TRAC-M-TR-13-050
14 June 2013

Mission Command Analysis Using Monte Carlo Tree Search

TRADOC Analysis Center - Monterey
700 Dyer Road
Monterey, California 93943-0692

This study cost the Department of Defense approximately $191,000 expended by TRAC in Fiscal Year 13. Prepared on 20130618 TRAC Project Code # 631

DISTRIBUTION STATEMENT: Approved for public release; distribution is unlimited. This determination was made on 14 June 2013.

This page intentionally left blank.

TRAC-M-TR-13-050
14 June 2013

Mission Command Analysis Using Monte Carlo Tree Search

MAJ Christopher Marks
Dr. Christian Darken
Dr. Arnie Buss
Dr. Kyle Lin
LTC Jonathan Alt

TRADOC Analysis Center - Monterey
700 Dyer Road
Monterey, California 93943-0692

Prepared by:

Christopher E. Marks
Analyst
TRAC-MTRY

Approved by:

Jonathan K. Alt
LTC, IN
Director, TRAC-MTRY

DISTRIBUTION STATEMENT: Approved for public release; distribution is unlimited. This determination was made on 14 June 2013.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing this collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden to Department of Defense, Washington Headquarters Services, Directorate for Information Operations and Reports (0704-0188), 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to any penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number. **PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS.**

1. REPORT DATE *(DD-MM-YYYY)*	**2. REPORT TYPE**	**3. DATES COVERED** *(From - To)*
14-06-2013	Technical Report	15-09-2012 --14-06-2013

4. TITLE AND SUBTITLE

Mission Command Analysis Using Monte Carlo Tree Search

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)

MAJ Christopher Marks, Dr. Christian Darken, Dr. Arnie Buss, Dr. Kyle Lin, LTC Jonathan Alt

5d. PROJECT NUMBER

631

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Training and Doctrine Command Analysis Center---Monterey
700 Dyer Road
Monterey, CA 93943-0692

8. PERFORMING ORGANIZATION REPORT NUMBER

TRAC-M-TR-13-050

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Training and Doctrine Command Analysis Center---Headquarters
255 Sedgwick Avenue
Fort Leavenworth, KS 66027-2345

10. SPONSOR/MONITOR'S ACRONYM(S)

TRAC

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION / AVAILABILITY STATEMENT

Approved for public release; distribution is unlimited.

13. SUPPLEMENTARY NOTES

14. ABSTRACT

In this project examine applications of Monte Carlo tree search, an artificial intelligence algorithm, in military simulation environments and assignment and scheduling problems with the goal of enhancing mission command analysis capabilities. We provide a review of recent literature on Monte Carlo tree search methods and then develop two algorithms that adapt the Monte Carlo tree search algorithm, traditionally applied to deterministic, fully observable games, to military simulations, which are typically stochastic and partially observable in nature. We develop, test, and comment on the results of two prototype implementations: one in a simple simulation environment with the objective of conserving friendly strength while depleting opposing forces, and the other focused on producing an optimal or near optimal assignment and schedule of aerial platforms against a set of missions with known values. Finally, we conclude by making recommendations for future implementations and applications in the COMBATXXI and JDAFS simulation environments, and suggest ways of addressing some of the computation challenges associated with Monte Carlo tree search and recursive simulation.

15. SUBJECT TERMS

Artificial Intelligence, Military Simulation, Monte Carlo Tree Search.

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON**
a. REPORT	**b. ABSTRACT**	**c. THIS PAGE**			MAJ Christopher E. Marks
Unclassified	Unclassified	Unclassified	SAR	93	**19b. TELEPHONE NUMBER** *(include area code)* (831) 656-3751

Table of Contents

List of Figures

List of Tables

List of Acronyms and Abbreviations

AI	Artificial Intelligence
ASC-U	Assignment and Scheduling Capability for Unmanned Aerial Vehicles
COA	Course of Action
COP	Combat Outpost
CVO	Constrained Value Optimization
DFP	Deployed Force Protection
FARP	Forward area refueling & rearming point
JDAFS	Joint Dynamic Allocation of Fires and Sensors
LRS	Launch and Recovery Site
MCTS	Monte Carlo Tree Search
MD	Mission Demand
MOVES	Modeling, Virtual Environments, and Simulation
NPS	Naval Postgraduate School
TRAC	Training and Doctrine Command Analysis Center
TRAC-MRO	Training and Doctrine Command Analysis Center Methods and Research Office
TRAC-MTRY	Training and Doctrine Command Analysis Center—Monterey
TRAC-WSMR	Training and Doctrine Command Analysis Center—White Sands Missile Range
XML	Extensible Markup Language

1. Introduction

The purpose of this technical report is to record the modeling approach, analysis, results and conclusions from the "Mission command analysis using Monte Carlo Tree Search" project.

1.1. Background

In the fall of 2012, the Training and Doctrine Command Analysis Center (TRAC) Methods and Research Office (TRAC-MRO) sponsored the Training and Doctrine Command Analysis Center—Monterey (TRAC-MTRY) to research the potential of employing Monte Carlo Tree Search (MTCS) methods in military simulations for the purpose of analyzing mission command. In this section we provide a brief overview of mission command, military simulations, and Monte Carlo Tree Search methods.

1.1.1. Mission command

From the Army field manual on operations, FM 3-0,

> *Mission command* is the conduct of military operations through decentralized execution based on mission orders. Network technology has affected the execution of mission command, and these network capabilities have been included in military simulation models. Mission command gives subordinates the greatest possible freedom of action. Commanders focus their orders on the purpose of the operation rather than on the details of how to perform assigned tasks. They delegate most decisions to subordinates. This minimizes detailed control and empowers subordinates' initiative. Mission command emphasizes timely decision making, understanding the higher commander's intent, and clearly identifying the subordinates' tasks necessary to achieve the desired end state. It improves subordinates' ability to act effectively in fluid, chaotic situations [7].

FM 3-0 establishes four elements of mission command:

- Commander's intent.
- Subordinate's initiative.
- Mission orders, which include—
  - A brief concept of operations.
  - Minimum necessary control measures.

- Resource allocation.

1.1.2. Military simulations

Military simulation environments generally attempt to model each of these elements. Subordinate units demonstrate initiative by reacting to simulation events as they occur, making decisions based on some rules, algorithms, or optimizations built into the simulation. Mission orders, including a concept of operations and controls, as well as resource allocation, are built into the simulated scenario. Finally, the commander's intent not only helps to establish the rules governing subordinate decision-making initiative, but also provides the criteria for measuring a unit's overall effectiveness.

Analysis of mission command and tactical decision making in military simulation environments is often limited to end-of-run metrics. From these metrics, it is not always apparent which decisions were *important*, i.e., they played a significant role in shaping the final outcome of the simulation, especially pertaining to how well a unit accomplished its mission and met the commander's intent.

1.1.2.1. Joint Dynamic Allocation of Fires and Sensors (JDAFS)

The Joint Dynamic Allocation of Fires and Sensors (JDAFS) simulation environment was originally developed by TRAC-MTRY in conjunction with the Naval Postgraduate School (NPS). It is a low-resolution, entity-level simulation designed to be rapidly configured and executed [3]. JDAFS employs event graphs, a discrete event simulation methodology, to simulate combat scenarios. It uses constrained value optimization models (CVOs) to allocate fires to targets and sensor platforms to potential areas of interest, and schedules events based on the outputs of these optimizers. It is relatively easy to create a scenario in extensible markup language (XML) or Microsoft™ Access and execute it in JDAFS, and scenarios generally run much faster than they would in other simulation environments.

1.1.2.2. Assignment and Scheduling Capability for Unmanned Aerial Vehicles (ASC-U)

The Assignment and Scheduling Capability for Unmanned Aerial Vehicles (ASC-U) is a scheduling tool that employs approximate dynamic programming methods, integer programming, and simulation. It uses the JDAFS simulation environment to model a set of aerial platforms including their capabilities, limitations, and logistical requirements. ASC-U finds a feasible assignment of these aerial platforms to a set of input mission demands using a rolling horizon approach. For a fixed time horizon, ASC-U solves an assignment optimization problem to find the best allocation of available aircraft to mission demands. It then begins executing this allocation in the simulation until the optimization interval (an input) has expired, at which point it runs the assignment optimization problem again and updates the platform allocation. The user defines the valuation rates of each sensor against each mission demand, which are multiplied by the platform's time covering the mission demand to determine the coefficients in the objective function of the

assignment optimization [1]. ASC-U has been used in several TRAC studies, including studies involving the scheduling of manned aerial assets.

1.1.2.3. COMBATXXI™

According to the COMBATXXI documentation [11], this simulation environment is a "joint, high-resolution, closed-form, stochastic, discrete event, entity level structure analytical combat simulation." It was developed and is supported by the Training and Doctrine Command Analysis Center—White Sands Missile Range (TRAC-WSMR) with collaboration from other centers and organization. It is designed to simulate brigade and lower echelon operations, but incorporates supporting assets from higher echelons. The purpose of COMBATXXI is to provide the needed resolution and modeling to support a variety of types of analyses.

1.1.3. Monte Carlo Tree Search

Monte Carlo tree search (MCTS) is a method that explores an action space in search of good or near-optimal solutions. The method attempts to balance *exploring* decision paths that have not yet been simulated and *exploiting* decision paths that are known to provide favorable results [2]. The goal of MCTS is to get a good idea of what the optimal or near-optimal decision paths are without having to enumerate and/or simulate the entire action space (an intractable alternative for all but the smallest game trees).

In military simulation environments, MCTS might be able to provide:

- New behavior models.
  - Decision theoretic style.
  - No engineering by an expert required.
  - Action choices explainable in mathematical terms.
- Component of naturalistic decision-making models ("mental simulation").
- Insight into decision spaces.
  - How many courses of action (COAs) are good?
  - How many COAs lead to disaster?

1.2. Problem Statement

To apply and evaluate Monte Carlo Tree Search (MCTS) methods to tactical decision situations in a simulated environment in order to expand mission command analysis capabilities.

1.3. Constraints, Limitations, and Assumptions

1.3.1. Constraints

- Complete by 01 July 2013.
- No contracting (constraint added 15 January 2013).

1.3.2. Limitations

- We will carry out all experimentation in TRAC-owned simulation environments for which we can obtain programmer support and in prototype implementations.
- Rewards functions will be defined by the study team.

1.3.3. Assumptions

- Conclusions will generalize to other scenarios in other simulations.

1.4. Project Team

- **Sponsor:** Mr. Paul Works, TRAC Research Director, MRO.
- **Project Lead:** MAJ Chris Marks (TRAC-MTRY).
- **Supporting Analyst:** LTC John Alt (TRAC-MTRY).
- **Supporting Analyst:** Mr. Blane Wilson (Training and Doctrine Command Analysis Center—White Sands Missile Range [TRAC-WSMR]).
- **NPS Faculty:** Dr. Chris Darken.
- **NPS Faculty:** Dr. Arnie Buss.
- **NPS Faculty:** Dr. Kyle Lin.
- **Programmer:** Mr. Terry Norbraten (NPS Modeling, Virtual Environments and Simulation [MOVES] Institute, JDAFS/ASC-U).

1.5. Methodology

We executed this project in four major efforts:

1. Initial implementation.
2. Scenario and algorithm refinement.
3. Development of a fires allocation algorithm (supporting effort).
4. Implementations.
5. Test and evaluation.

The initial implementation effort included the original problem definition work, literature review, and decisions on how to "discretize" a state space and action space in a military simulation environment. These initial ideas would be built into a simple, prototype MCTS implementation. In parallel with the MCTS prototype, the team developed a fires allocation algorithm for future integration with MCTS implementations. With insights gleaned from the prototype MCTS implementation, the team moved on to refine the MCTS application scenario and algorithm for use in a military simulation environments. After considering different environments and in response to changes in programmer resources available, the team turned its focus on ASC-U as the initial simulation environment in which to implement MCTS. This effort included building the code necessary to implement MCTS into a military simulation environment. Finally, once the team had a working MCTS implementation, the team conducted some basic tests to determine the value of MCTS as a tool to analyze mission command in future applications. Figure 1.1 depicts the methodology for this project.

1.6. Organization

This chapter gives the background and problem definition for this project. In chapter 2, we give a more in-depth overview of the MCTS algorithm and some of its existing applications, and conclude with some of the challenges involved with implementing MCTS into military simulations. Chapter 3 provides our answers to these challenges, including modifications to existing MCTS methods and a fires allocation algorithm developed as a supporting effort in this project. We also present our prototype implementation along with the insights it provides in chapter 3. In chapter 4 we describe the application of MCTS in ASC-U and discuss the results of this implementation. Chapter 5 develops implementation algorithms and scenarios for JDAFS and COMBATXXI for future research efforts. Finally, in chapter 6 we conclude this report with a summary of results and insights into the applicability of MCTS in military simulation.

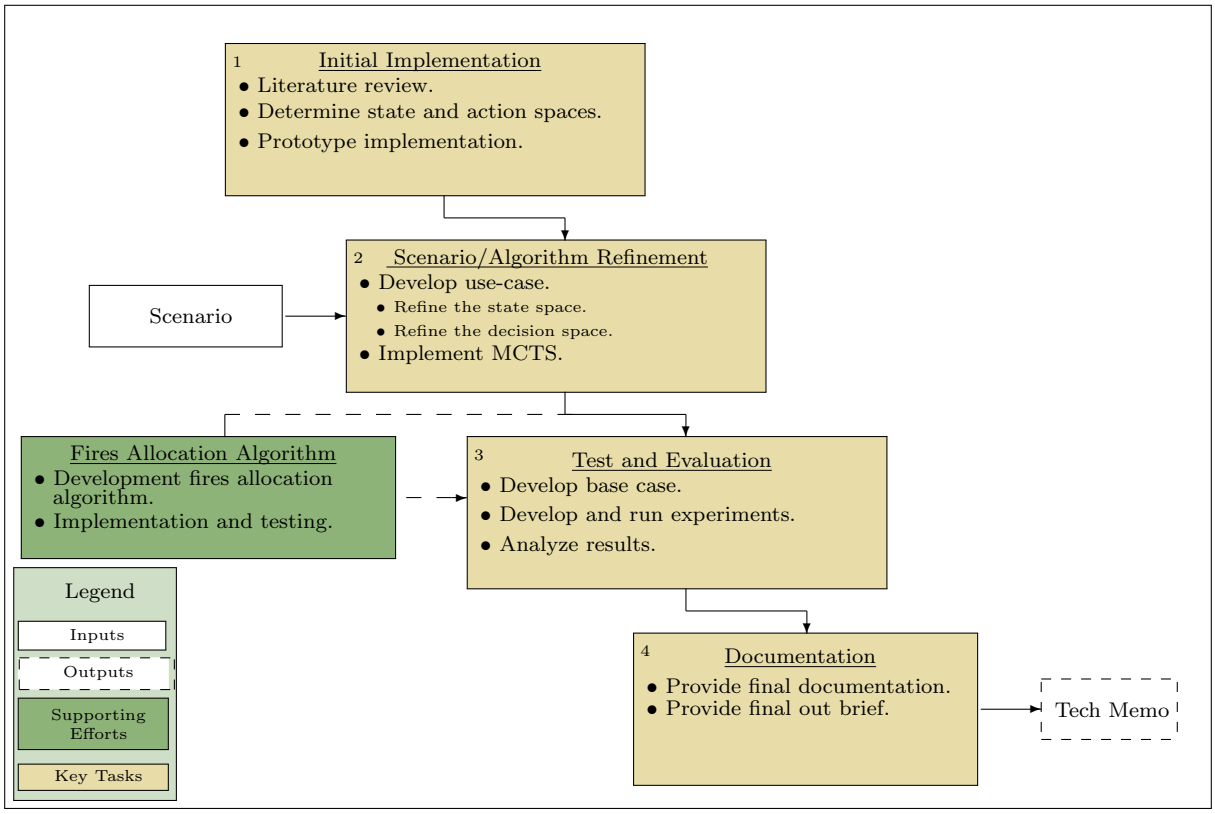

Figure 1.1: MCTS project methodology.

2. Monte Carlo Tree Search

In this chapter we provide a more detailed look at the MCTS algorithm and its current applications and discuss it major challenges with implementation into a military simulation environment.

2.1. MCTS Algorithm

Monte Carlo Tree Search is a method that applies sampling and simulation to find optimal or near optimal decisions in a discrete, finite action space. MCTS has had significant impact in the artificial intelligence (AI) community, primarily because of its success at identifying good moves in computer "Go," a virtual version of an ancient Chinese board game. Previously, computer Go had presented a difficult challenge to the AI community. Like chess and many other board games, the huge state and action space makes it impossible to enumerate the entire game tree. However, unlike chess and many other board games it is very difficult to place objective value measures on certain moves or board positions in Go. This characteristic made it very difficult for AI algorithm developers to build a virtual Go player capable of matching human competence. The Monte Carlo Tree Search method provided the answer to this difficult challenge, providing a decision algorithm capable of competing against the some of the best human Go players on small boards (MCTS is still not competitive on the standard 19×19 board) [2].

Monte Carlo Tree Search has several characteristics that have made it successful in computer Go, "bandit" problems, and other AI applications:

- MCTS does not require any domain-specific knowledge of state or decision values (e.g., position scoring functions) [2].
- It is an "any time" algorithm [2]. It can always provide an answer, but the answer will get better with more computation time.
- The algorithm theoretically converges to the optimal policy.
- MCTS balances *exploration* (sampling decision trajectories that have been sampled a relatively low number of times in previous iterations) and *exploitation* (sampling decision trajectories that have produced favorable results in previous iterations), resulting in an intelligent sampling from the game tree [2] .

The algorithm consists of two policies: the *tree* policy and the *default* (or *simulation*) policy.

2.1.1. The Tree Policy

The tree policy constructs and intelligently samples from the *game tree*, a network representation that depicts *states* as nodes and *decisions* (or *actions*) as directional arcs. Starting at the root node and proceeding down a decision path, the tree policy iteratively selects child nodes (i.e., actions leading to new states) according to a rule until it reaches a previously unexplored state.

One tree search rule for that has created effective tree policies is the maximum upper confidence bound for trees (UCT) [2]. For a child node j,

$$UCT_j = \overline{X}_j + C_p\sqrt{\frac{2\ln n}{n_j}},$$

where

$\overline{X}_j$ is the average reward resulting from the collection of decision paths sampled in previous iterations that include child node j,

C_p is a nonnegative constant,

n is the number decision paths sampled in previous iterations that include the current node, and

n_j is the number of decision paths sampled in previous iterations that have included child node j.

A maximum UCT tree policy computes the UCT values for all of a node's children and then selects the child node with the maximum value (if the maximum is shared by more than one child, then one of these children is selected arbitrarily). The expression for UCT demonstrates how the tree policy balances exploration and exploitation. The first term, $\overline{X}_j$, will favor exploitation of nodes that have consistently returned high reward values in previous iterations. By containing the number of visits for the child node in the denominator, the second term encourages exploration and will be larger for lower values of n_j. In a specific case, if a child node j has never been visited the second term causes the $UCT_j \to \infty$. Finally, the value for C_p is defined by the user. Browne et al. suggest $C_p = \frac{1}{\sqrt{2}}$ and point to theoretical research supporting the use of this value [2].

2.1.2. The Default Policy

Once the tree policy has selected an unexplored node, MCTS executes the default policy. Beginning at the state represented by the selected node, the algorithm simulates a decision path according to user-defined rules until the game ends. In many applications, the rule governing the default policy has been to iteratively select a decision at random from the available action space [2]. The goal of the default policy is to quickly produce a possible outcome of the game that could result from the state selected by the tree policy. Once the default policy reaches a terminal state, the resulting final outcome is scored according to

its value using a user-defined reward function. The MCTS algorithm then updates the values for n_j and $\overline{X}_j$ for all nodes j in the decision path identified by the tree policy (nodes and arcs found by the default policy are not added to the MCTS game tree or updated with any values).

2.1.3. Algorithm Example

For simplicity, we present an example scenario in which a decision maker must execute a sequence of binary decisions. In the first iteration, the algorithm begins at the root node. Because neither of the children have been explored, the UCT for each child node is ∞. The algorithm randomly picks one of these children (e.g., node "1") and then executes the default policy to obtain a reward value (e.g., 0.75). This process is illustrated in figure 2.1. The algorithm then updates the average reward ($\overline{X}_j$) and the number of visits (n_j) for the node selected by the tree policy, used to compute UCT values in the next iteration.

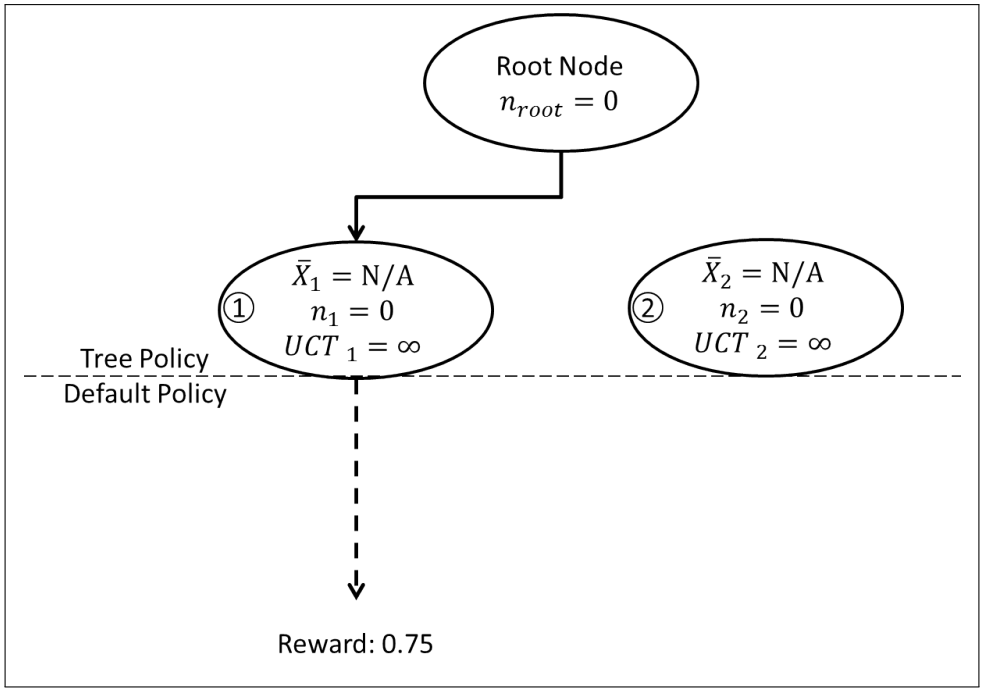

Figure 2.1: MCTS example, first iteration.

In the second iteration again begins at the root node. Note that of the two child nodes, one now has a finite UCT (node 1, selected during the previous iteration) while the other UCT is still infinite. Seeking always to maximize UCT, the tree policy selects the other child node (node "2") for this iteration and then executes the default policy to obtain a reward value. See figure 2.2 for an illustration of this iteration. Once the default policy terminates and findes the reward value, the algorithm updates the average reward and number of visits for node 2. While it is not strictly necessary to store the overall average reward for the root node, we do keep track of the total number of iterations in order to compute UCT values for nodes 1 and 2.

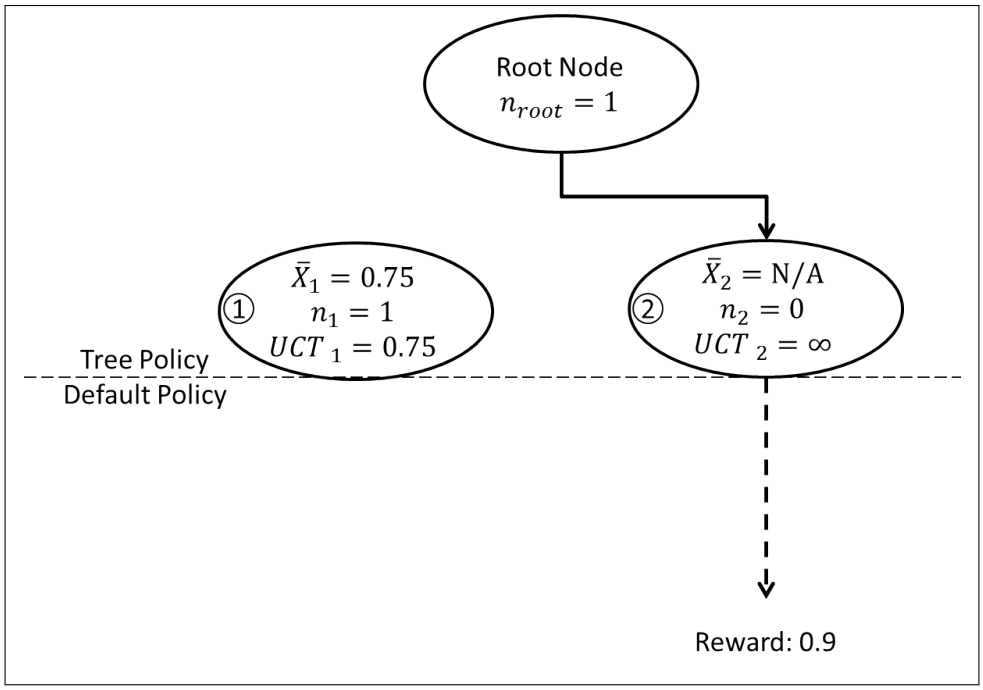

Figure 2.2: MCTS example, second iteration.

In the third iteration, the algorithm again compares the UCT values for nodes 1 and 2, and selects the node with the higher value. In our example (see figure 2.3), node 2 has the higher UCT in this iteration. The tree policy selects node 2, but does not terminate because this node has already been visited at least once. Instead the tree policy again compares the UCT values for the child nodes, this time for nodes "3" and "4." These nodes represent states that can be reached by actions from the state represented by node 2. Because neither node 2 child has been selected by the tree policy in previous algorithm iterations, their UCT values are infinite. The tree policy selects one of them (e.g., node "4") randomly and, now that it has identified an unexplored node, the algorithm executes the default policy to obtain a reward value (figure 2.3). This reward value is used to update the average reward for node 4 and node 2. The algorithm also updates the number of visits for the root node, node 2, and node 4.

Beginning as always from the root node, the tree policy selects node 1 in the fourth iteration because its UCT is higher than that of node 2. Here we note that even though node 2 has a higher average reward, because it has been visited more than node 1 it has a lower UCT value (see figure 2.4). The tree policy proceeds by selecting one of node 1's child nodes at random because neither of them have been previously visited. From this selected node, the algorithm runs the default policy and updates the selected nodes' average reward value and number of visits.

The algorithm proceeds in this fashion until some user defined stopping criterion is reached. This criterion could be a maximum number of iterations, a fixed amount of computational time, or a maximum tree size or memory allocation. Once the algorithm

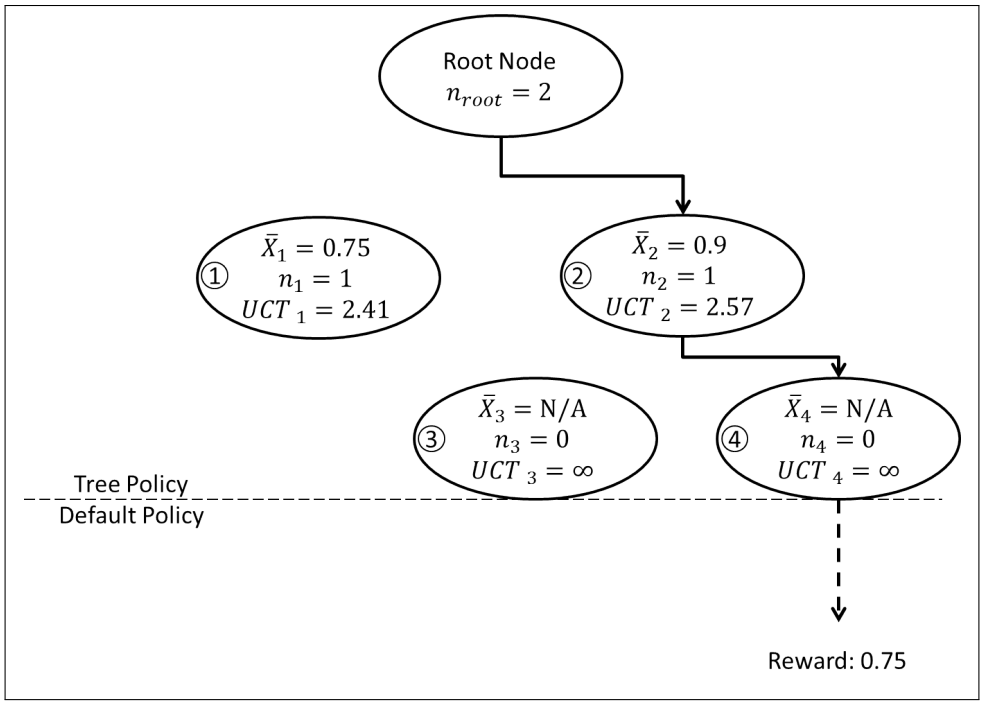

Figure 2.3: MCTS example, third iteration.

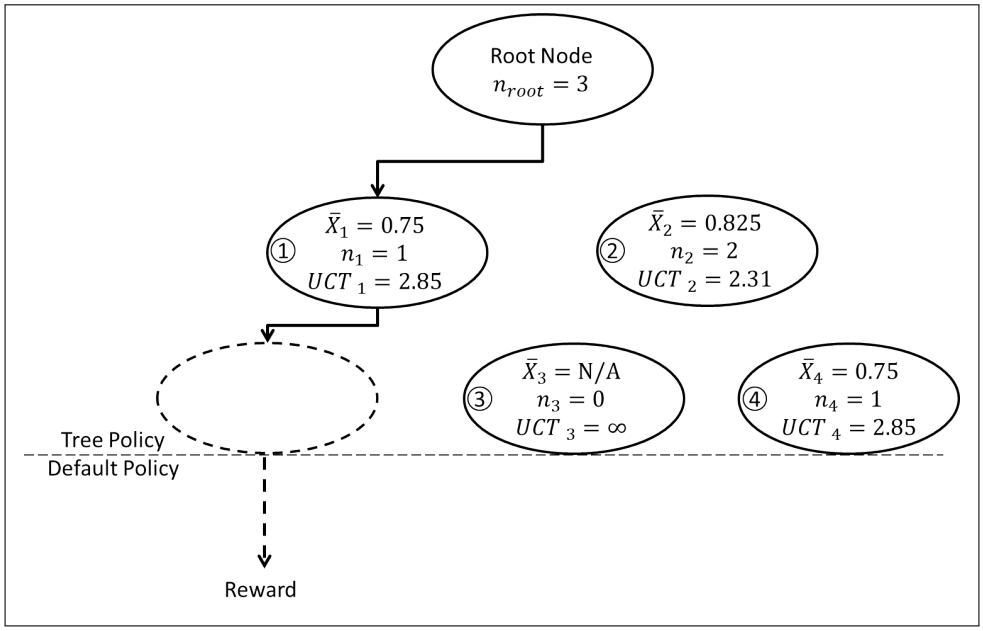

Figure 2.4: MCTS example, fourth iteration.

terminates, it returns the best decision available from the root node, often determined by the best average reward among the child nodes.

In summary, the MCTS algorithm executes as follows:

1. While **stopping criteria** have not been met:
   (a) Set **current node** equal to *root node*.
   (b) Execute the tree policy. While **current node** is not unexplored:
      i. Compute UCT_j for all **current node** child nodes j.
      ii. Arbitrarily pick one node from the subset of child nodes $\{j' | UCT_{j'} = \max_j [UCT_j]\}$.
      iii. Set **current node** equal to this node.
      iv. Store the **current node** identification index in **this iteration set**.
   (c) Execute the default policy. While **current node** is not a terminal state:
      i. Arbitrarily select an action from the set of possible actions for the **current node**.
      ii. Set **current node** equal to the resulting child node.
   (d) Compute the reward value associated with terminal state **current node**.
   (e) Update values of $\overline{X}_j$ and n_j for all nodes in **this iteration set**. Update $\overline{X}_j$ so that this average includes the reward value from this iteration. Increment n_j by one.
2. Select the action from the root node that results in the highest average reward, $\overline{X}_j$, from the root node's children and terminate.

Note that the output of the algorithm is an action to take based on the results of the tree search. While in the example provided above we choose the action that has shown the highest average reward, other selection criteria exists. See Browne et al. for discussion of alternate selection criteria [2]. It is also important to note that the above algorithm runs each time a decision is required in a scenario.

2.2. The challenges in implementing MCTS into military simulations

Monte Carlo Tree Search methods have been employed in a wide variety of games including Connect4, Scrabble, Chess, and Go. Each of these games enables all players to have immediate, perfect state information. Furthermore, the players' actions result in deterministic outcomes, or state transitions.

Unfortunately, these characteristics, desirable from a MCTS implementation perspective, are typically not characteristics of military simulation environments or military operations.

Perfect information on locations and activities of enemy elements, on the locations and activities of friendly elements, or on the terrain are usually not made available to any entities when constructing and running scenarios in military simulation environments. Also, actions, such as firing at a target, do not always produce the same results in military simulation environments, with the consequences determined by the outputs of pseudo-random number generators.

Successfully implementing MCTS into a military simulation in an applicable way requires adapting the algorithm so that it can account for:

- Imperfect state information.
- Stochastic outcomes.

Browne et al. mention several adaptations of MCTS tailored for stochastic, imperfect state information applications [2]. *Determinization* forms the basis for the method we develop, which we present in the next chapter [2, p. 13].

Another challenge in implementing MCTS into military simulations is the need to reproduce and simulate forward from specific intermediate simulation states in the search tree. The complicated and dynamic nature of military simulation environments makes this requirement an nontrivial obstacle to successful implementation. Gilmer Jr and Sullivan discuss these difficulties and introduces several techniques for "multi-trajectory simulation", which they define as "[allowing] random events in a simulation to generate multiple trajectories and [explicitly] managing the set of trajectories" [5]. Gilmer Jr and Sullivan further discuss the benefits of "recursive" simulation, in which decision entities within the simulation have the ability to create and run their own instances of the simulation, using the results to guide their actions [6]. Implementing MCTS in a military simulation environment is a logical extension to the recursive simulation work done by Gilmer Jr and Sullivan [6].

In addition to the multi-trajectory simulation techniques provided in Gilmer Jr and Sullivan, Gilmer Jr and Sullivan, we propose the following methods to overcome some of the challenges to producing multiple replications of an intermediate simulation state:

- Re-start the simulation from the initial state with the same random seed and make the same sequence of decisions in order to arrive at the same state. The benefit to using this method is that it requires storing very little information and allows MCTS to run as a control almost entirely external to the simulation. One drawback, however, is that in large scenarios and complex simulation environments, re-running the simulation multiple times from the initial state through several decision points in order to reproduce a specific state can consume a large amount of time. Also, this method somewhat overrides the stochastic nature of the simulation, making outcomes always deterministic.

- Abstract the MCTS state space from the simulation state. For the purpose of the MCTS algorithm, define the state only as the information required by the decision-maker in choosing a course of action. We can build the search tree on this information set, instead of on the actual simulation state, and modify the UCT formula slightly to account for stochastic outcomes. This concept, known as *Information Set UCT (ISUCT)*, is presented and discussed in Browne et al. [2, p. 13].

3. Algorithm and Prototype Development

3.1. MCTS in Stochastic Games with Imperfect State Information

In this section we introduce the partial observation-UCT (PO-UCT) algorithm, an adaptation of the MCTS UCT algorithm described above and based on the notion of "determinization" [2]. The purpose of this adaptation is to apply MCTS methods to stochastic games with partially observed states while staying as close to the original algorithm as possible. Because stochasticity can generally be considered to be due to lack of full observability (e.g. the die rolls appear random because we lack full knowledge of the state of the dice as they are rolled), we can answer both of these issues (stochasticity and imperfect state information) with a single modification to the algorithm.

We modify the structure of the search tree to contain two different types of nodes, "chance" and "player" in alternating layers. Chance nodes contain as children various states that are possible after a given sequence of moves after considering sources of stochasticity, i.e., the lack of full knowledge of the initial state, or the stochastic effects of player actions. Player nodes correspond to nodes from the original UCT algorithm. They contain child chance nodes that represent the possible states that can occur if a player makes a specific action choice.

The root of the tree is a chance node, with subsequent generations of nodes alternating between player nodes and chance nodes. Chance nodes contain the action a player took to get to them, except for the root, which has a "null" action, as it represents a point in time before any of the player actions under consideration have been taken. Player nodes contain a single, fully-specified game state.

The PO-UCT algorithm computes an estimate of the best action for a particular player to take at a particular point in time. To initiate the algorithm, a set $\{s_1, s_2, \ldots, s_k\}$ of fully-specified states that are each consistent with the information set of this player must be provided.

If j is a search tree node, we assume that the following functions are available: $type(j)$, which is either "chance" or "player", $p(j)$, which returns the parent node of j, $s(j)$, which returns the state stored at the node assuming the node type is "player", $a(j)$, which returns the action stored at the node assuming the node type is "chance", and $f(s, a)$, which returns independent samples of the state distribution resulting from taking action a in state s. n_j is the number of visits to the node (including traversals on the way to a child), and q_j the sum of all quality values produced for this node or its children. $\Delta(j, p)$ is the component of the reward vector Δ associated with on-move player p at node j. It is unclear to us under what circumstances dependence upon j is needed, but it has been retained for consistency with Browne et al. [2]. For example, in a two player zero sum game, Δ would always be a two-element vector of values summing to zero, e.g. $[x \ -x]$. If

is s a state, there is a function $A(s)$ that returns the set of actions that are possible for the on-move player. C_p is $1/\sqrt{2}$.

3.1.1. The k-Sample Partial Observation MCTS Algorithm

In this section we provide pseudo-code for our k-sample partial observation (kSPO) algorithm. This algorithm produces a fixed, user-defined number (k) of "player" nodes from every chance node in the manner described in the preceding paragraphs.

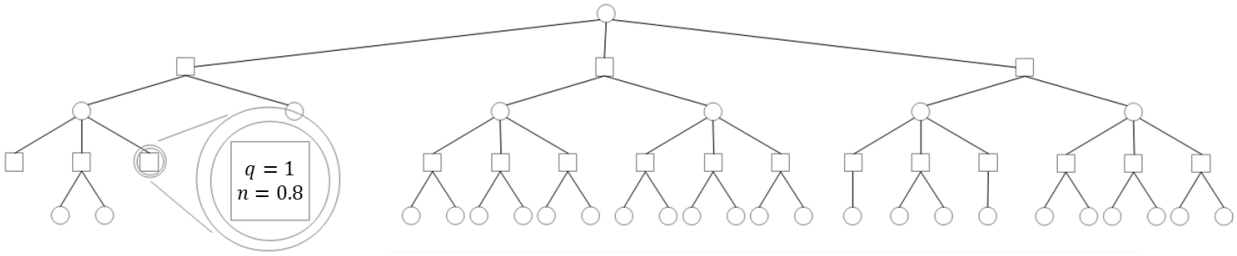

Figure 3.1: An example kSPO MCTS search tree.

function KSPO-UCTSearch($s_1, s_2, \ldots, s_k$)
 create root chance node j_c and player node children $j_{p1}, j_{p2}, \ldots, j_{pk}$ with $s(j_{pi}) = s_i$
 while within computational budget **do**
  $j_l \leftarrow$ TreePolicy(j_c)
  $\Delta \leftarrow$ DefaultPolicy($s(j_l)$)
  Backup(j_l, Δ)
 return a(BestChild($j_c, 0$))

function TreePolicy(j)
 while j is nonterminal **do**
  if *type*(j) is "chance"
   $j \leftarrow$ ChanceNodePolicy(j)
  if *type*(j) is "player"
   $j \leftarrow$ PlayerNodePolicy(j)
 return j

function ChanceNodePolicy(j)
 if j is not *root* and has less than k children
  $j \leftarrow$ TryAction($a(j), p(j), j$)
 else
  j is a node with minimum n_j
 return j

function PlayerNodePolicy(j)
 if an action is untried
  choose a in untried actions from $A(s(j))$
  create child chance node j_c with $a(j_c) = a$

$j \leftarrow$ TryAction(a, j, j_c)
else
$j \leftarrow$ BestChild(j, C_p)
return j

function TryAction(a, j_p, j_c)
add a new child j to j_c with $s(j) = f(s(j_p), a)$
return j

function BestChild(j, Δ) (Note: this function comes directly from [2].)
return $\underset{j' \in \{\text{children of j}\}}{\arg\max} \frac{q_{j'}}{n_{j'}} + C_p \sqrt{\frac{2 \ln n_j}{n_{j'}}}$

function DefaultPolicy(s) (Note: this function comes directly from [2].)
while s is non-terminal **do**
choose $a \in A(s)$ uniformly at random
$s \leftarrow f(s, a)$

function Backup(j_l, Δ) (Note: this function comes directly from [2].)
while j is not *null* **do**
$n_j \leftarrow n_j + 1$
$q_j \leftarrow q_j + \Delta(j, p)$
$j \leftarrow$ parent of j

Figure 3.1 depicts a search tree that could result from using this algorithm. The circular nodes in this tree represent chance nodes, while the square nodes represent player nodes. The game represented has a binary action space, as indicated by the two arcs descending from each player node. Each player node represents a complete simulation state and has associated number of visits (n) and total utility (q) values. Figure 3.1 also depicts example values for a single player node in the tree. We note that the tree is not symmetric as the search tree explores and exploits according to the max-UCT rule.

The kSPO MCTS algorithm outlined in this section is unlikely to converge to an optimal solution for most games, because a finite number of samples, however large, is not sufficient to represent an arbitrary probability distribution. The algorithm presented in the next section continually adds additional independent random samples while simultaneously exploring the existing samples.

3.1.2. The Incrementing Sample Partial Observation (ISPO) Algorithm

In this section we provide pseudo-code for our incrementing sample partial observation (ISPO) algorithm. This algorithm produces an incrementally growing number (k) of "player" nodes from every chance node. Let the function $g()$ sample from the distribution of fully-specified states that are each consistent with the information set of the player

initially on move. All functions except those provided below are the same as for kSPO-UCT Algorithm (see paragraph 3.1.1).

function ISPO-UCTSearch($s_1, s_2, \ldots, s_k$)
 generate a state sample $s_1 = g()$
 create root chance node j_c and player node child j_{p1} with $s(v_{p1}) = s_1$
 while within computational budget **do**
  $j_l \leftarrow$ TreePolicy(j_c)
  $\Delta \leftarrow$ DefaultPolicy($s(j_l)$)
  Backup(j_l, Δ)
 return a(BestChild($j_c, 0$))

function ChanceNodePolicy(j)
 n is the number of children of j
 m is the minimum n_{j_c} for all j_c, children of j
 if $m \geq n$
  if j is root
   $j \leftarrow$ NewInitialState(j)
  else
   $j \leftarrow$ TryAction($a(j), p(j), j$)
 else
  $j \leftarrow$ an arbitrary child of j with $n_j = m$
 return j

function NewInitialState(j_c)
 add a new child j to j_c with $s(j) = g()$

return j

Unlike the kSPO MCTS algorithm, the ISPO algorithm continues to create new player nodes from chance nodes as the tree is explored and populated. As the number of iterations increases, the ISPO algorithm continues to sample from the distribution of fully-specified states that are consistent with the information set for the player on move, as well as from the distribution of outcomes possible for each action. In the limit, the algorithm will exhaustively sample from each distribution, giving the ISPO algorithm the same convergence properties as the UCT algorithm developed in Browne et al. [2]. Computationally, however, the kSPO algorithm is much easier to implement and much less resource intensive to run. Therefore, we based our prototype implementation on the kSPO algorithm.

3.2. Prototype Implementation

The purpose of the prototype implementation was to confirm that our kSPO MCTS algorithm would perform in a simple simulation and to uncover any obstacles that we might have to overcome to complete more complicated implementations. In this implementation, we set $k = 3$ samples per chance node. We built a small scenario in which four blue entities face off against ten red immobile red entities on a square, featureless terrain for 30 seconds. See figure 3.2 for a representation of this scenario.

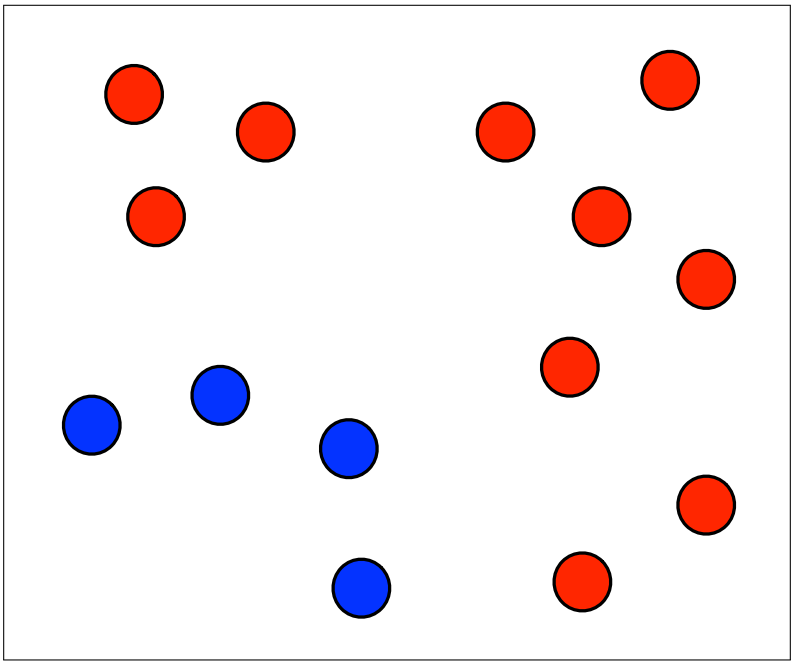

Figure 3.2: Illustration of the prototype scenario.

3.2.1. Action Space

The blue units have two pre-programmed actions:

- **Concentrate**. Units move towards the nearest point on a circle of fixed radius about their center of mass (see figure 3.3. They do not stop to shoot when concentrating.
- **Hunt**. Units move towards the nearest known red position. When they get in range they stop and repeatedly shoot until they or the target are dead. If no red position is known, units move towards a point on the terrain sampled from a uniform distribution. See figure 3.4 for an illustration of this action.

The blue commander picks one of these two options at 0, 10, and 20 seconds into the run.

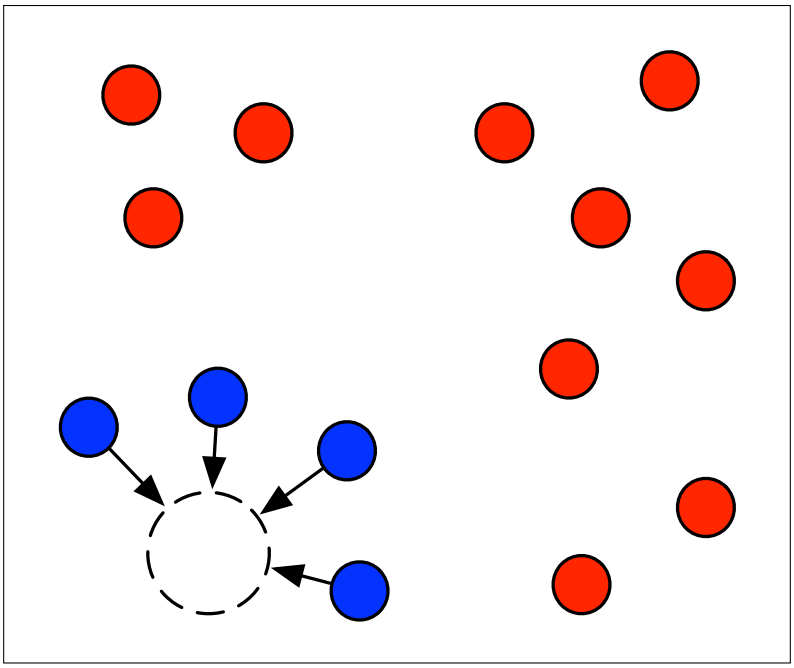

Figure 3.3: Illustration of the "concentrate" action.

3.2.2. Reward Function

The reward function for this scenario is the normalized difference in blue and red forces remaining, i.e., for any terminal node j,

$$\Delta(j) = \frac{10 + x_{s(j)} - y_{s(j)}}{14},$$

where $x_{s(j)}$ is the number of blue forces alive in state $s(j)$ and $y_{s(j)}$ is the number of red forces alive in state $s(j)$.

3.2.3. Tree Policy

For the tree policy we use the kSPO-UCT policy described in paragraph 3.1.1. We set $k = 3$ samples per chance node. Considering the $a = 2$ actions available at each of $p = 3$ decision points, the size of the fully-enumerated search tree is

$$k(ak)^p = 3(6)^3 = 648 \text{ nodes.}$$

The tree depicted in figure 3.1 gives an example of what this search tree might look like during one iteration of the kSPO MCTS algorithm. In the first iteration, according to the kSPO MCTS algorithm, the program will generate three complete simulation states based on the partial information available to the blue forces. The MCTS algorithm proceeds from these three fully defined states.

3.2.4. Default Policy

We employ a default policy that chooses actions at random for this implementation.

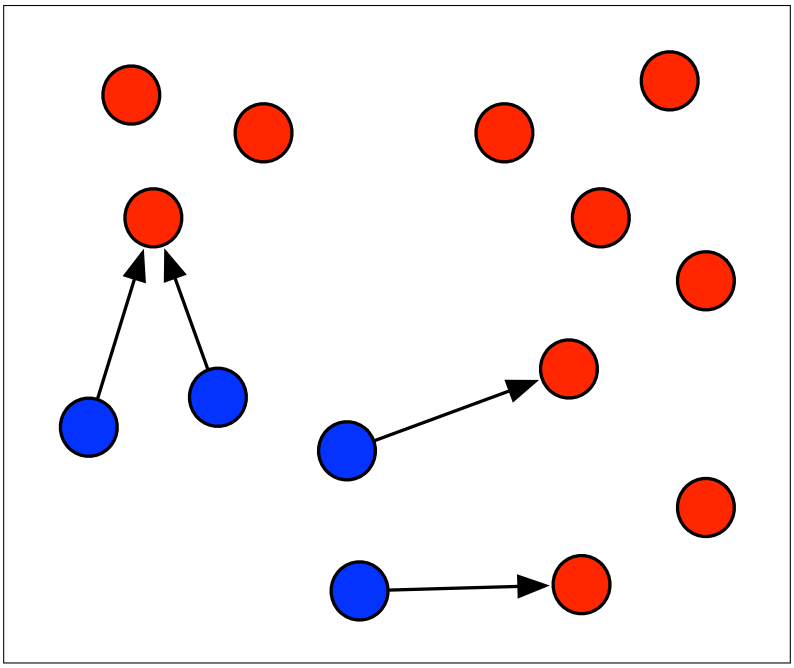

Figure 3.4: Illustration of the "hunt" action.

3.2.5. Results

The team built the prototype in Javascript. The prototype ran in a few seconds and produced a searchable tree in according to the kSPO MCTS algorithm. From the prototype implementation, the team recorded the following lessons learned:

- When implementing MCTS into partially observable games, we must be able to produce a fully-specified state based on available information. In this case, implementing the kSPO MCTS algorithm requires building a commander's situation sampling function. Representing the commander's subjective distribution of the current state of the battle is a difficult research problem that is beyond the scope of this project.
- Reconstructing an intermediate state in simulation is challenging. It proved easier to rerun the simulation using the same initial random seed and then to make the same sequence of decisions that resulted in the desired state.
- When using the kSPO MCTS algorithm, it is important that the k-sized sample of outcomes (children) from every chance node provide a representative sample of the outcomes likely to occur in the simulation. While beyond the initial scope of this project, follow-on implementations of MCTS in military simulation would benefit by employing importance sampling heuristics.
- The size of the search tree in the kSPO MCTS algorithm can quickly make using this method intractable. Effective employment of this algorithm requires the user to limit

the size of the action space ($a = 2$, a constant in the prototype), the number of decision points ($p = 3$ in the prototype), and the number of entities making decisions (in the prototype, a single blue "commander" made the decision to execute either a "hunt" or "concentrate" policy). This does not necessarily preclude the use of MCTS in larger, more complicated scenarios, provided the user scopes the application of MCTS to a few "important" decision points.

3.3. Fires Allocation Algorithm

In addition to MCTS algorithm development and implementation, the research team also developed a fires allocation algorithm in order to implement along with the MCTS algorithm. The intended uses of the fires allocation algorithm include the following:

- As a method of pruning low-payoff options in the MCTS search tree, in the event the tree becomes too large.
- As a benchmark behavior policy to use in evaluating MCTS in a tactical firefight scenario.
- To serve as an alternative reward function, instead of using end-of-run metrics to build the reward value. Using the state value produced by the fires allocation algorithm in this manner can save time by producing a reward value for a state without having to run the simulation to completion.
- As a stand-alone behavior to implement in JDAFS or COMBATXXI.

3.3.1. Algorithm Notation and Assumptions

This algorithm assumes two forces, A, and B, are in direct fire contact. Force A consists of m elements, which we denote as set S_A, and force B has n elements, denoted by the set S_B. Each element $i \in (A \cup B)$ has one or more weapon systems from set W available to engage opposing elements. Let $W_i \subset W$ be the set of weapon systems element i has available to engage opposing elements.

If element $i \in S_A$ fires weapon $k \in W_i$ at element $j \in S_B$, then we assume in this algorithm that the time it takes i to kill j follows an exponential distribution with rate λ_{ijk}. If element $j \in S_B$ fires at element $i \in S_A$ using weapon $k \in W_j$, we assume that the time it takes for j to kill i is exponentially distributed with rate θ_{jik}. For this algorithm we approximate these kill rates with the following expressions:

$$\lambda_{ijk} = \frac{p_{ijk}}{\Delta_{ijk}},$$

$$\theta_{jik} = \frac{p_{jik}}{\Delta_{jik}}.$$

In this expression, p_{ijk} (or p_{jik}) is the probability i kills j (or j kills i) by firing a single round using weapon system k at j (i) in the simulation. This probability is typically a function of the situational factors such as engagement range, weather conditions, etc., that might be modeled in the simulation environment, and is employed by the simulation to determine whether an element dies each time it is engaged by another element. The quantities Δ_{ijk} and Δ_{jik} represent the periods of the engagement, i.e., $\frac{1}{\Delta_{ijk}}$ gives i's rate of fire for weapon system k at target j and $\frac{1}{\Delta_{jik}}$ gives j's rate of fire for weapon system k at target i.

3.3.1.1. Forces in Contact as a Markov Process

At any time point, the state of the overall engagement can be delineated by (S'_A, S'_B), with $S'_A \subset S_A$ being the set of A's remaining units, and $S'_B \subset S_B$ the set of B's remaining units. Now we define a state value function, $V(S'_A, S'_B)$, defined for states (S'_A, S'_B) such that $S'_A \subset S_A$ and $S'_B \subset S_B$, with the exception of state $(\emptyset, \emptyset)$. This function returns the probability that force A wins the overall engagement by killing all elements of force B, conditioned on starting from the state defined by (S'_A, S'_B), and represents the value of that state to force A. Note that the value of the state from the perspective of force B is $1 - V(S'_A, S'_B)$. In this function we implicitly assume that the outcome probabilities can be completely determined from the current state information, independent of past states, putting this model in the class of Markov games. Also, note that $V(\emptyset, S'_B) = 0$ for $S'_B \neq \emptyset$ and $V(S'_A, \emptyset) = 1$, for $S'_A \neq \emptyset$. We assume that overall engagement has zero probability of entering state $(\emptyset, \emptyset)$, and leave function V undefined for this state.

3.3.1.2. Fire Allocations

We now define a fire allocation for each force in state (S_A, S_B). For all $i \in S_A, j \in S_B, k \in W_i$, we define

$$x_{ijk} = \begin{cases} 1, & \text{if } A'\text{s unit } i \text{ fires at } B'\text{s unit } j \text{ using weapon } k, \\ 0, & \text{otherwise.} \end{cases}$$

Also, for all $i \in S_A, j \in S_B, k \in W_j$, we define

$$y_{jik} = \begin{cases} 1, & \text{if } B'\text{s unit } j \text{ fires at } A'\text{s unit } i \text{ using weapon } k, \\ 0, & \text{otherwise.} \end{cases}$$

The set of fire allocations for A is

$$\Pi_A = \left\{ \mathbf{x} = [x_{ijk}] : x_{ijk} \in \{0,1\}, i \in S_A, j \in S_B, k \in W_i; \sum_{j \in S_B} \sum_{k \in W_i} x_{ijk} = 1 \ \forall i \in S_A \right\}.$$

If each element $i \in S_A$ has only one weapon system, the number of unique fire allocations available to A is n^m. Assuming each element has at least one weapon system, this value provides a lower bound on the number of fire allocations available to A. We can also establish an upper bound as $n^{\alpha m}$ if we know the maximum number of weapons, α, that any

element $i \in S_A$ possesses. Similarly, we define the set of fire allocations for B:

$$\Pi_B = \left\{ \mathbf{y} = [y_{jik}] : y_{jik} \in \{0,1\}, j \in S_B, i \in S_A, k \in W_j; \sum_{i \in S_A} \sum_{k \in W_j} y_{jik} = 1 \; \forall j \in S_B \right\}.$$

Again by assuming each element $j \in S_B$ has at least one weapon system and no more than β weapon systems, we can establish m^n as a lower bound and $m^{\beta n}$ as an upper bound on the number of fire allocations available to B. We implicitly assume in these definitions that each element can only engage one target with one weapon system in a fire allocation.

3.3.1.3. The Value of a Fire Allocation

In state (S_A, S_B), given A's fire allocation $\mathbf{x} \in \Pi_A$, let

$$\Lambda_j(\mathbf{x}) = \sum_{i \in S_A} \sum_{k \in W_i} x_{ijk} \lambda_{ijk}$$

denote the rate at which element $j \in S_B$ gets killed. In other words, the amount of time it takes for A to kill B's unit j follows an exponential distribution with rate $\Lambda_j(\mathbf{x})$, if A uses fire allocation $\mathbf{x}$. Likewise, given B's fire allocation $\mathbf{y} \in \Pi_B$ in state (S_A, S_B), let

$$\Theta_i(\mathbf{y}) = \sum_{j \in S_A} \sum_{k \in W_j} y_{jik} \theta_{jik}$$

denote the rate at which element $i \in S_A$ gets killed.

Given state (S_A, S_B), in which A employs fire allocation $\mathbf{x}$ and B employs fire allocation $\mathbf{y}$, we can express the probability that A will eventually win using the law of total probability and the special characteristics of exponential distributions as

$$f(\mathbf{x}, \mathbf{y}) = \frac{\sum_{j \in S_B} \Lambda_j(\mathbf{x}) V(S_A, S_B \setminus \{j\}) + \sum_{i \in S_A} \Theta_i(\mathbf{y}) V(S_A \setminus \{i\}, S_B)}{\sum_{j \in S_B} \Lambda_j(\mathbf{x}) + \sum_{i \in S_A} \Theta_i(\mathbf{y})}$$

If both A and B employ the optimal fire allocation, then the value of the two-person zero-sum game defined by the preceding payoff function is then $V(S_A, S_B)$—the state value function for (S_A, S_B).

Dr. Kyle Lin proves that—in the case in which each element has a single weapon system—the two-person zero-sum game defined by the preceding payoff function has a saddle point [9]. In other words, pure strategies suffice to be optimal, and one does not need to consider mixed strategies. He goes on to give necessary and sufficient conditions for an optimal fire allocation $(\mathbf{x}^\star, \mathbf{y}^\star)$, and then provides an efficient way of finding the optimal fires allocations. This algorithm generalizes for cases when one or more element has multiple weapon systems simply by adding another third dimension to the inputs as we have demonstrated in the preceding paragraphs.

3.3.2. Algorithm Development

Lin gives the following necessary and sufficient conditions for $(\mathbf{x}', \mathbf{y}')$ to be an optimal fire allocation, and a real number v' to be the state value, in state (S_A, S_B):

C1. $\mathbf{x}'$ maximizes $\sum_{j \in S_B} \Lambda_j(\mathbf{x}) \cdot (V(S_A, S_B \setminus \{j\}) - v')$.

C2. $\mathbf{y}'$ minimizes $\sum_{i \in S_A} \Theta_i(\mathbf{y}) \cdot (v' - V(S_A \setminus \{i\}, S_B))$.

C3. $f(\mathbf{x}', \mathbf{y}') = v'$.

For a full proof, see Lin [9].

We now present the algorithm for finding the optimal fires allocation. Lin proves that this algorithm always converges to the optimal fires allocation in a finite number of steps [9].

Algorithm

1. Pick v arbitrarily in $[0, 1]$.
2. For $v \in [0, 1]$, define

$$\hat{\mathbf{x}}(v) \equiv \arg\max_{\mathbf{x}} \sum_{j \in S_B} \Lambda_j(\mathbf{x}) \cdot (V(S_A, S_B \setminus \{j\}) - v),$$
$$\hat{\mathbf{y}}(v) \equiv \arg\min_{\mathbf{y}} \sum_{i \in S_A} \Theta_i(\mathbf{y}) \cdot (V(S_A \setminus \{i\}, S_B) - v).$$

   In case of a tie, break it arbitrarily. Next, compute

$$T(v) \equiv f(\hat{\mathbf{x}}(v), \hat{\mathbf{y}}(v)) = \frac{\sum_{j \in S_B} \Lambda_j(\hat{\mathbf{x}}(v)) V(S_A, S_B \setminus \{j\}) + \sum_{i \in S_A} \Theta_i(\hat{\mathbf{y}}(v)) V(S_A \setminus \{i\}, S_B)}{\sum_{j \in S_B} \Lambda_j(\hat{\mathbf{x}}(v)) + \sum_{i \in S_A} \Theta_i(\hat{\mathbf{y}}(v))}.$$

3. If $T(v) = v$, then v is the value of the game and $(\hat{\mathbf{x}}(v), \hat{\mathbf{y}}(v))$ is a saddle point. If $T(v) \neq v$, then update $v \leftarrow T(v)$, and go to step 2.

MATLAB™ code implementing this algorithm is included in Appendix A. This code includes a clever way of choosing a starting value for v in order to lessen the time it takes to converge. Appendix A also contains an example MATLAB™ script that employs the fires allocation algorithm.

This page intentionally left blank.

4. MCTS Implementation in ASC-U

The implementation of MCTS is the assignment and scheduling problem has a few notable differences from the other implementations in this project. First, ASC-U is a deterministic model. Second, ASC-U does not represent an enemy or force-on-force operations. Its purpose is to schedule aerial assets against a known set of mission areas that occur in time and space, making it a fully-observable model. In these two characteristics, a MCTS implementation in ASC-U is more resembles its historical uses in games such as computer Go than its other implementations in military simulation environments.

4.1. The MCTS Algorithm for ASC-U

In this section we develop an application of the MCTS algorithm for implementation in an assignment and scheduling problem.

4.1.1. Action Space

Each mission area constitutes a decision point, and decisions are made in sequential order. In the event that two or more mission areas begin at the same time, the algorithm arbitrarily orders them. For each mission area (i.e., at each decision point) the algorithm can assign any single available platform-sensor package combination to the mission demand, or leave the mission unfulfilled. Platform-sensor package combinations are available if the equipment is not already obligated to support other mission areas.

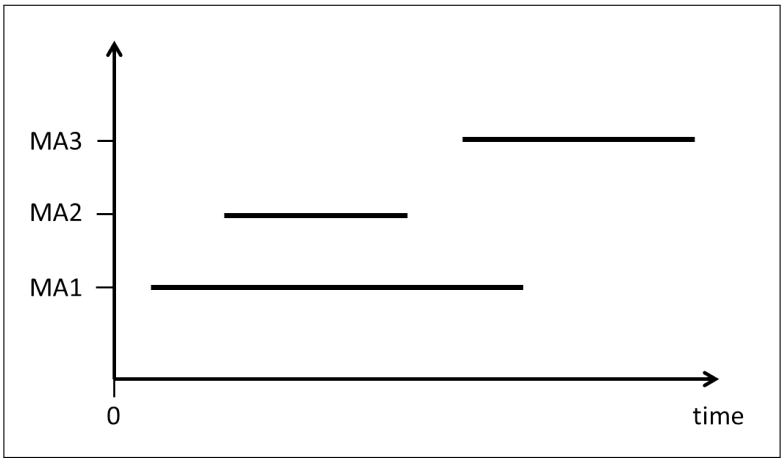

Figure 4.1: A three mission-area example, depicted in time.

For example, suppose we have a simple scenario with the three mission areas depicted in figure 4.1. From the figure, we can see that a platform assigned to mission area 1 (MA1) cannot execute MA2 or MA3 because these missions begin while MA1 is still in open. A platform assigned to MA2, however, is available to execute MA3, assuming certain reset

and refueling constraints are met, because these missions do not overlap. Assuming there are two platforms available, the first three generations of the resulting search tree are depicted in figure 4.2. Note that platform availability in the third generation only depends on actions taken from the root node. Even though the nodes represent the same decision point with the same platforms available, they still represent different *states* because the differences in platform allocations to the first two mission areas might result in different utility values. Finally, at each decision point there is always the option to leave the mission unfulfilled. These decisions are represented by the rightmost arcs coming from each node.

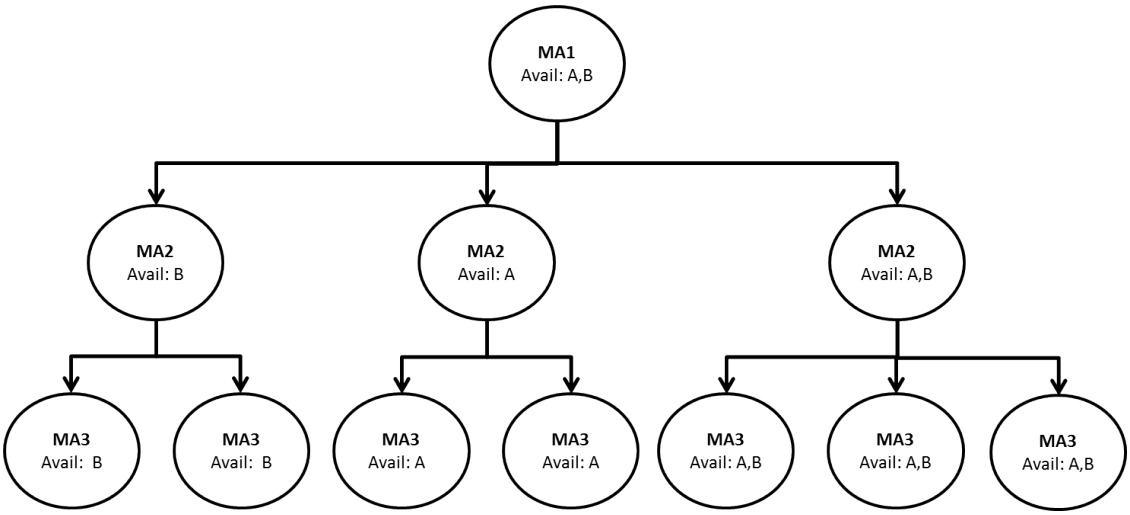

Figure 4.2: Given two platforms and the mission areas depicted in figure 4.1, a depiction of the resulting search tree.

4.1.2. Reward Function

The reward function for a specific assignment of platforms to missions is the objective function ASC-U uses in its optimization formulation [1]:

$$\Delta(j) = \sum_{s \in S} \sum_{a \in A} c_{s,a} X_{s,a},$$

where

S	is the set of sensor packages,
A	is the set of mission areas,
$c_{s,a}$	is the reward for assigning sensor package $s \in S$ to mission area $a \in A$, and
$X_{s,a}$	is a binary variable that takes value 1 if sensor package $s \in S$ is assigned to mission area $a \in A$, 0 otherwise.

4.1.3. Search Algorithm

We consider several variations of the UCT search algorithm:

- The UCT algorithm as described in Browne et al. [2]. The disadvantage to this

algorithm is that it does not employ domain specific knowledge, and therefore might waste resources investigating assignments that do not make sense.

- A modification to the UCT algorithm that includes a term for the immediate reward associated with a particular assignment and does not approach ∞ for unvisited nodes. This algorithm incorporates domain specific knowledge and might spend more time investigating better actions. However, it will require a correction that forces it to also explore the option of no assignment.
- The UCT algorithm employed on a pruned tree. This algorithm will determine the best $n-1$ assignments possible at each node, based on the immediate reward for the assignment, where n is specified in advance by the user. Then, it will construct the search tree using only these possible assignments, and the option of no assignment, as the action space. By ignoring low payoff options this method will limit the size of the search tree to no more than n actions available from each node. There is risk, however, in overlooking an assignment with a relatively low immediate payoff that leads to a better overall assignment.

4.1.4. Default Algorithm

We consider several possible default algorithms:

- Random assignments.
- Greedy assignments. This algorithm selects the best assignment available from the pool of potential assignments based on immediate reward.
- Epsilon-greedy algorithm [8]. This algorithm either selects the action with the highest immediate payoff or selects an action at random according to the outcome of a Bernoulli trial. It represents a compromise between the pure random and pure greedy assignments.
- Pruned random assignments. This algorithm will randomly select one action from the set consisting of:
  - The best $n-1$ assignments possible, based on the immediate reward, where n is specified in advance by the user.
  - The no-assignment option.

4.2. ASC-U Prototype in MATLAB

We created a prototype MCTS implementation in MATLAB aimed at solving the assignment and scheduling problem. This ASC-U prototype does not have all of the same capabilities as ASC-U, noteably:

- It does not allow for multiple launch and recovery sites (LRSs). Instead, it uses a single LRS located at (0,0).
- It does not allow for LRS repositioning. The LRS is assumed static.
- It does not model any constraints pertaining to ground control stations.
- It does not allow for platforms to move from one mission assignment directly to another. This is probably the most significant shortcoming of the current MCTS ASC-U prototype in MATLAB.
- It does not model forward area refueling & rearming points (FARPs).

With additional time, each of these limitations, as well as others not listed, could be addressed. Specifically, updating the MATLAB prototype to enable platforms to move from one mission directly to another would require only a modest programming effort. However, even with these limitations the MCTS ASC-U prototype in MATLAB enables meaningful analysis and comparison with ASC-U results. Unlike ASC-U, the MCTS ASC-U prototype in MATLAB does not rely on a rolling horizon heuristic, potentially enabling for more efficient solutions to similar assignment problems.

The MATLAB functions and script files comprising this prototype are provided with documentation in Appendix B.

4.2.1. Prototype Algorithm and Data Requirements

As a tree policy, the MATLAB prototype employs the standard UCT algorithm described in Browne et al. [2]. It uses random assignments as the default policy. It requires three arrays of data as input:

- A `platform` array, which provides the operational endurance, operating speed, and reset time for each platform-sensor package combination. The reset time is the amount of time the platform must remain on the ground after landing following a mission.
- A `mission` array, which provides mission number, mission start time, mission duration, mission location (x and y coordinates), and value rate for each mission.
- A `performance` array, which provides a mission effectiveness coefficient for each platform-sensor package against each mission.

The state space and action space follow the definitions provided above. If a platform is available to take off and cover any part of a mission, the prototype creates a node for it for exploration. The reward for a specific platform-sensor package combination, p, assigned to a single mission, m is:

$$(t_d - t_a) \cdot R_m \cdot V_{p,m},$$

where
t_d is the time the platform-sensor package departs the mission area,
t_a is the time the platform-sensor package arrives at the mission area,
R_m is the value rate for mission m, and
$V_{p,m}$ is the effectiveness coefficient for assigning platform p to mission m.

Total reward is simply the sum of all individual assignment rewards. Finally, we scale the value for the constant C_p in the UCT expression (see section 2.1.1) to account for total possible reward given unlimited assets:

$$C_p = \frac{1}{\sqrt{2}} \left(\sum_{m \in A} d_m \cdot R_m \right)$$

where
A is the set of missions (or mission areas),
d_m is the duration of mission m, and
R_m is the value rate for mission m.

4.3. Testing and Results

We conduct two simple tests using the prototype. Both tests serve the purpose of verifying that the algorithm produces a feasible schedule that makes sense based on the inputs. The first test is an attempt to reproduce the results from ASC-U testing documented in *Modeling Updates in Support of Armed Aerial Scout* [4]. The second test provides a more complicated input data set and observes the effect of changing the number of MCTS iterations prior to making a decision.

4.3.1. First test: Verification Using a Simple Input Scenario

The first verification test consists of three cases, each a simple scenario consisting of one platform-sensor combination and four missions. The purpose of this test is to verify the functionality of the ASC-U prototype and ensure that the output makes sense.

4.3.1.1. Input Data

For these three cases, the `mission` and `performance` input arrays remain constant; only the `platform` speed and operational endurance parameters are varied. Table 4.1 shows the input `platform` vector for each of the three cases. Tables 4.2 and 4.3 show the set of missions and performance coefficients respectively for the three cases. As the performance coefficients indicate, for this test we assume that the platform has the capability required to accomplish each mission with 100% effectiveness. Finally, table 4.4 gives the value for the constant, C_p, in the UCT expression and the number of iterations, B used to define the computational budget, or stopping criteria, for the MCTS algorithm. As mentioned previously, we scale the value of C_p for this scenario by multiplying by total mission value.

The computational budget input, B, is the number of times the algorithm iterates through the tree policy, the default policy, and the back-propagation steps of the MCTS algorithm before returning a decision and moving on to the next decision in the process (see sections 2.1.3, 3.1.1, and 3.1.2).

Case	Platform	Endurance	Speed	Reset
1	1	4.00	250.00	8.00
2	1	4.00	3125.00	8.00
3	1	10.00	3125.00	0.50

Table 4.1: Platform inputs for cases 1-3.

Mission #	Start time	Duration	x coordinate	y coordinate	Value rate
1	0.00	16.00	100.00	0.00	1.00
2	0.00	4.00	500.00	0.00	60.00
3	5.00	4.00	900.00	0.00	20.00
4	12.00	4.00	900.00	0.00	100.00

Table 4.2: Mission inputs for cases 1-3.

Platform	Mission #1	Mission #2	Mission #3	Mission #4
1	1.00	1.00	1.00	1.00

Table 4.3: Performance inputs for cases 1-3.

Parameter	Value
C_p	$\frac{1}{\sqrt{2}}\left(\sum_{m \in A} d_m R_m\right) = \frac{736}{\sqrt{2}}$
B	100

Table 4.4: UCT constant and computational budget for cases 1-3.

4.3.1.2. Results

The ASC-U prototype using MCTS in MATLAB performed as one might expect against the simple data inputs for this test. In each case, the schedule produced clearly represents a good course of action within the feasible set of alternatives. Table 4.5 provides the objective value and runtime for each of the three cases. The following paragraphs give the results for each of the three input cases that comprise the first test.

	Objective Value	runtime (seconds)
Case 1	6.4000	1.3420
Case 2	563.2000	1.3880
Case 3	595.2000	1.2790

Table 4.5: Objective values and runtimes for cases 1-3.

Case 1

For case 1, the ASC-U protoype produced the schedule shown in table 4.6 and depicted in figure 4.3. In the figure, the red horizontal lines represent the four missions and the black lines represent the platform traveling to the mission area, remaining on station (and accruing value), and then returning to the LRS. We find that the platform executes the lowest priority mission until its operational endurance expires. Then the platform lands, resets and and returns to the lowest priority mission for a second sortie. This case served as the base case to test the functionality of the MATLAB ASC-U implementation. The platform does not perform any of the higher priority missions because they are out of its range as constrained by its speed and operational endurance. These results were not consistent with the ASC-U results, most likely because our platform speed is set too slow.

Platform	Mission #	Take-off	Arrive	Depart	Land
1	1	-0.40	0.00	3.20	3.60
1	1	11.60	12.00	15.20	15.60

Table 4.6: Schedule produced for case 1.

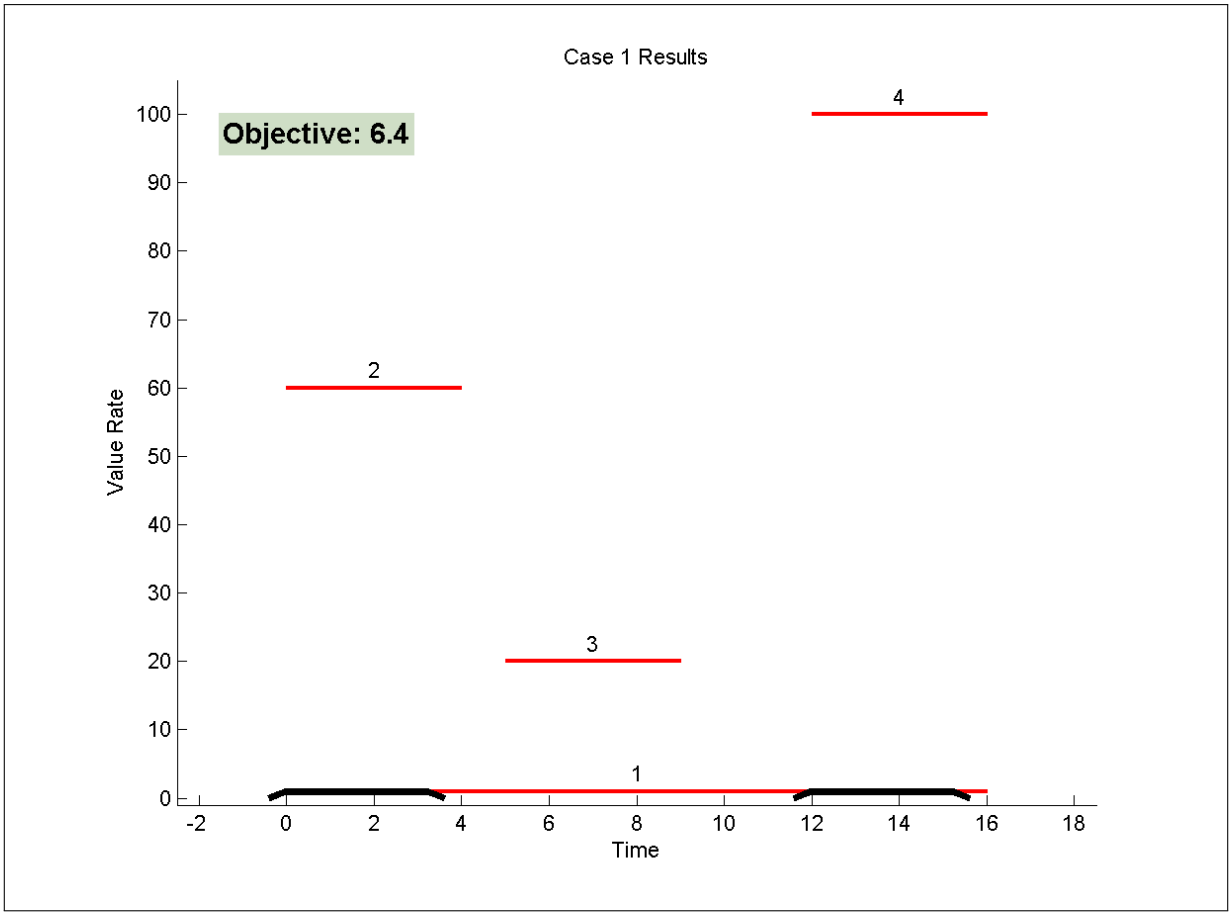

Figure 4.3: Visual depiction of schedule for case 1.

Case 2

In case 2, we take the input scenario from case 1 and increase the platform speed in order to enable it to reach all missions. We choose a platform speed 3125 (see table 4.1 in order to make the travel time to mission #2 take 0.16 time units, consistent with the outputs in previous testing (see Buss et al. [4]). The prototype produced a schedule resulting in a total mission value score of 563.2. While this objective does not match the objective value from the previous tests in ASC-U, the schedule produced (see table 4.7) is essentially the same as that produced in the first scenario in Buss et al. [4]. Figure 4.4 depicts this schedule. Because of the eight-unit reset time, the schedule produced leaves mission #3 unfilled.

Platform	Mission #	Take-off	Arrive	Depart	Land
1	2	-0.16	0.00	3.68	3.84
1	4	11.84	12.13	15.55	15.84

Table 4.7: Schedule produced for case 2.

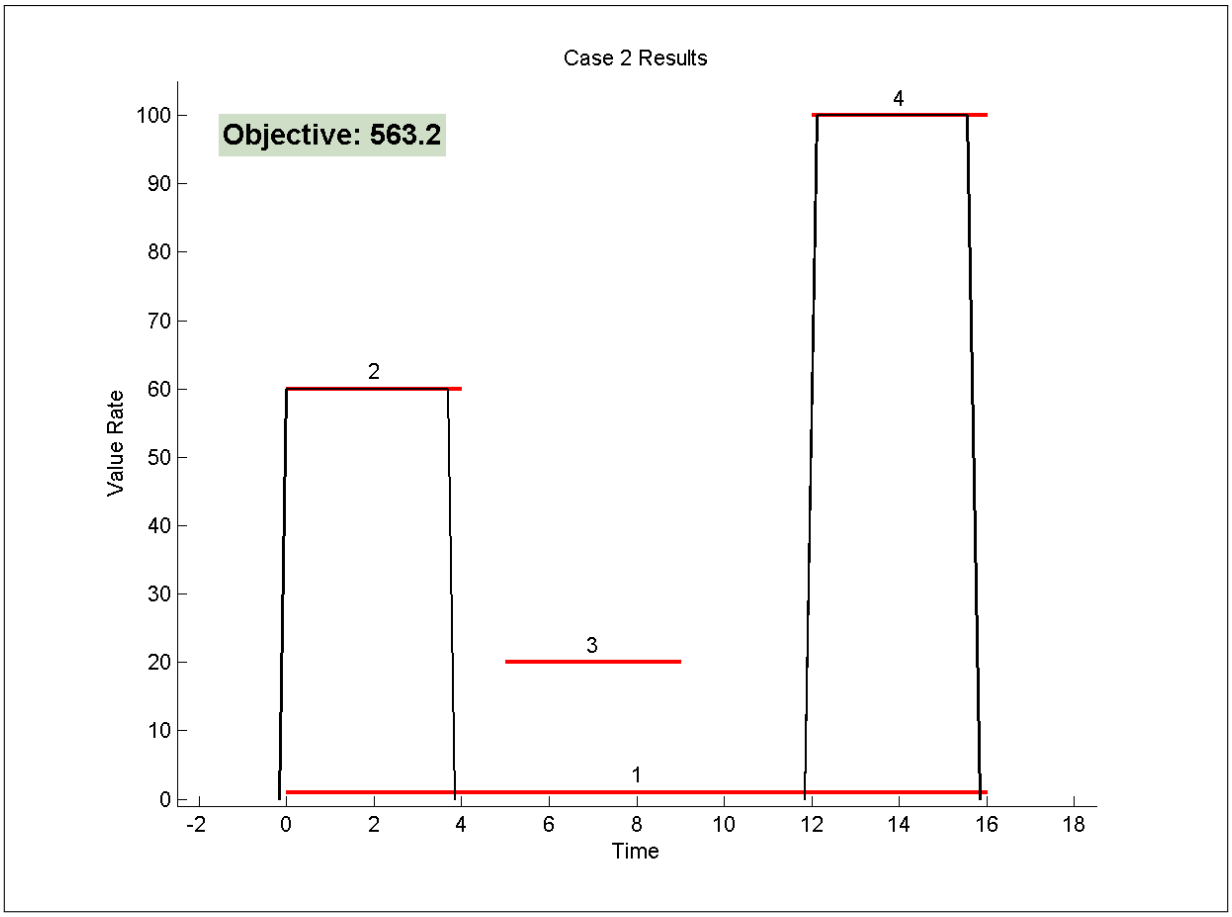

Figure 4.4: Visual depiction of schedule for case 2.

Case 3

In this case we increase the operational endurance of the aircraft from 4 time units to 10. We also decrease the reset time to 0.5 time units, diverging from the previous tests, which held the reset time constant, at eight time units. The reason for our divergence from the documented ASC-U tests (see Buss et al. [4]) is that the remainder of the previous tests involved platforms conducting multiple missions without landing, a capability which does not yet exist in the MATLAB prototype. Making the reset time very small, on the other hand, enables many interesting courses of action for assignment and scheduling.

The results highlight both the capabilities and the limitations of this current implementation. The schedule given in table 4.8 and depicted in figure 4.5 show a mission assignment that makes sense and covers all of the high priority missions, but it is also clear that we could do better with the following capabilities:

1. Enable platforms finishing missions to re-route to other missions, instead of returning to the LRS.
2. Enable platforms on low priority missions to be re-assigned to higher priority missions.
3. Enable platforms on low priority missions to depart the mission early in order to reset and become available to fill a high priority mission.

Platform	Mission #	Take-off	Arrive	Depart	Land
1	2	-0.16	0.00	4.00	4.16
1	3	4.71	5.00	9.00	9.29
1	4	11.71	12.00	16.00	16.29

Table 4.8: Schedule produced for case 3.

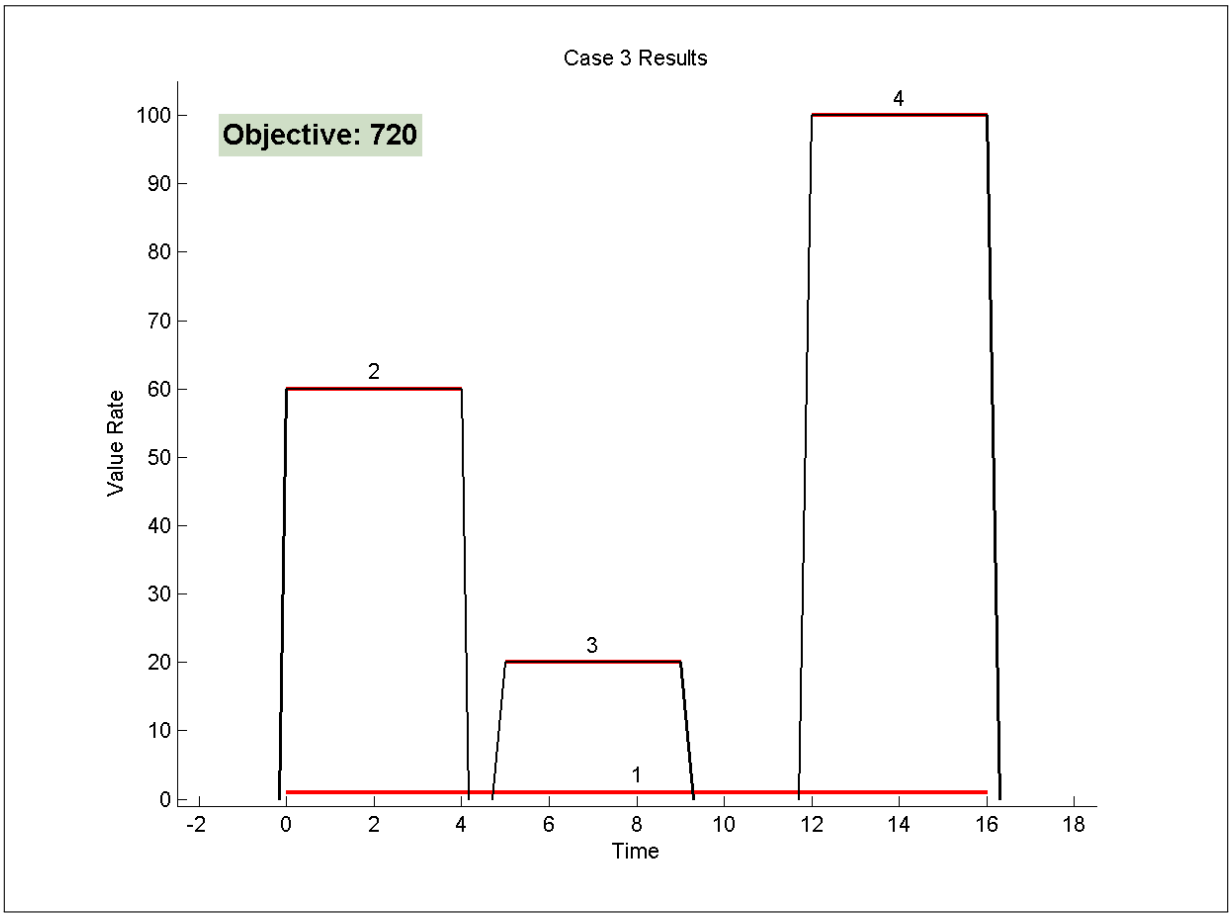

Figure 4.5: Visual depiction of schedule for case 3.

4.3.2. Second Test: Examining the Trade-off Between Performance and Computational Time

In this test we produce a more complicated set of input data and examine the impact of changing the computational budget (four levels) on the performance and efficiency of the ASC-U prototype.

4.3.2.1. Input Data

Except for the computational budget, B, the input data for the four cases (cases 4-7) that comprise this experiment is constant. Table 4.9 shows the number of MCTS iterations per decision for each case in this test. The test scenario consists of 9 platforms to be scheduled against 16 missions occurring over a 25 time-unit period. The `platform` input array is given in table 4.10, the `mission` input array is in table 4.11, and the `performance` input array is in table 4.12. Most of this data was generated using pseudo-random number generators to achieve various distributions. Generating data from pseudo-random numbers introduces the possibility of having many solutions to the problem that have very different schedules but result in similar, near-optimal objective values. In other words, by using random data we make the problem difficult to solve to optimality.

The value for C_p is again scaled by the total mission value in the scenario:

$$C_p = \frac{1}{\sqrt{2}} \left(\sum_{m \in A} d_m R_m \right) = 5.9395.$$

Case	B
4	10
5	100
6	1000
7	10000

Table 4.9: Computational budget for cases 4-7.

Platform	Endurance	Speed	Reset
1	2.18	13.24	5.58
2	1.93	16.33	3.75
3	2.85	11.28	4.65
4	2.48	10.31	4.42
5	1.55	12.18	4.71
6	1.53	14.38	3.59
7	2.03	16.85	2.13
8	1.54	11.38	3.55
9	1.65	14.99	4.46

Table 4.10: Platform inputs for cases 4-7.

Mission #	Start time	Duration	x coordinate	y coordinate	Value rate
1	0.58	2.18	4.65	4.99	0.64
2	0.68	1.55	3.72	5.86	0.28
3	0.86	0.63	2.38	6.00	0.11
4	1.45	1.33	1.46	2.70	0.28
5	3.42	1.06	0.41	4.90	0.23
6	9.67	2.94	0.98	7.77	0.62
7	11.29	0.56	1.17	0.80	0.99
8	11.51	1.02	1.38	0.64	0.75
9	12.37	3.18	1.72	8.51	0.02
10	13.74	1.24	7.02	9.05	0.67
11	14.03	1.99	7.80	4.30	0.00
12	14.89	1.63	2.57	3.91	0.16
13	15.73	2.13	8.40	5.80	0.18
14	17.60	1.43	9.66	3.29	0.21
15	18.98	0.86	1.32	7.85	0.70
16	20.24	2.90	2.51	2.77	0.10

Table 4.11: Mission inputs for case 4-7.

Platform	Mission #1	Mission #2	Mission #3	Mission #4	Mission #5	Mission #6	Mission #7	Mission #8
1	0.99	0.31	0.54	0.22	0.04	0.76	0.92	0.25
2	0.79	0.04	0.06	0.43	0.19	0.85	0.58	0.44
3	0.51	0.22	0.47	0.51	0.44	0.19	0.30	0.27
4	0.34	0.03	0.19	0.55	0.11	0.63	0.96	0.87
5	0.32	1.00	0.63	0.51	0.22	0.16	0.39	0.51
6	0.81	0.24	0.45	0.73	0.54	0.45	0.22	0.38
7	0.22	0.48	0.09	0.75	0.35	0.81	0.76	0.24
8	1.00	0.94	0.98	0.96	0.47	0.64	0.82	0.08
9	0.05	0.31	0.73	0.38	0.64	0.17	0.66	0.83

Platform	Mission #1	Mission #2	Mission #3	Mission #4	Mission #5	Mission #6	Mission #7	Mission #8
1	0.72	0.21	0.03	0.77	0.79	0.66	0.55	0.37
2	0.66	0.65	0.12	0.13	0.63	0.26	0.55	0.68
3	0.07	0.39	0.65	0.97	0.88	0.94	0.25	0.21
4	0.55	0.18	0.99	0.03	0.13	0.92	0.25	0.56
5	0.27	0.15	0.94	0.13	0.30	0.76	0.94	0.92
6	0.84	0.08	0.70	0.03	0.81	0.86	0.53	0.12
7	0.70	0.37	0.82	0.70	0.55	0.34	0.63	0.18
8	0.95	0.14	0.62	0.86	0.21	0.76	1.00	0.09
9	0.98	0.09	0.03	0.48	0.77	0.93	0.87	0.25

Table 4.12: Performance inputs for case 4-7.

4.3.2.2. Results

Due to the stochastic nature of MCTS, because of the algorithm's use of pseudo-random number generators to break ties and to make decisions when executing the default policy, the results of each case varied, sometimes greatly, for trials initiated with different random seeds. In order to capture the effect of changing the computational budget, we ran 100 replications for each case and observed the distributions of the results. As one might expect, the replications of case 7, with 10000 MCTS iterations per decision, took longer than the replications of the other cases, all of which made decisions using a smaller computational budget. However, the distribution of the 100 objective value results for case 7 had a higher mean and lower variance than the other cases, so the additional computational time resulted in improved performance.

Figure 4.6 shows box plots representing the objective value distributions for cases 4-7. The figure also provides mean objective value ($\bar{z}_i$), median objective value ($\tilde{z}_i$), objective value standard deviation (s_{z_i}), and mean runtime ($\bar{t}_i$), for each case consisting of $i = 10, 100, 1000, 10000$ iterations per decision. We see that case 4, with only 10 MCTS iterations per decision, performs much less consistently and, on average, somewhat worse than the cases using 100, 1000, and 10000 iterations per decision. We confirm the general trend that, by increasing computational budget, we get results that are more consistent and, on average, better. We also significantly increase the algorithm's runtime. However, we note that the performance of the upper quantiles for cases 5 and 6 (with 100 and 1000 iterations, respectively) is comparable to the performance of case 7 (with 10000 iterations), with only a fraction of the computational cost. This is especially true when looking at the upper quantiles in each distribution; it appears that the upper quantiles might converge at a faster rate than the mean, median, or lower quantiles. These results imply that it might be a better use of time to run multiple replications of the MCTS-ASCU prototype with a low number of iterations than to run one replication with a large number of iterations.

We now provide example coverage plots, similar to those for cases 1-3, the four cases that comprise this test. These four plots depict one of the 100 iterations that form the box plots in figure 4.6. Table 4.13 shows the mission assignments and objective values for these four example cases, for comparison purposes.

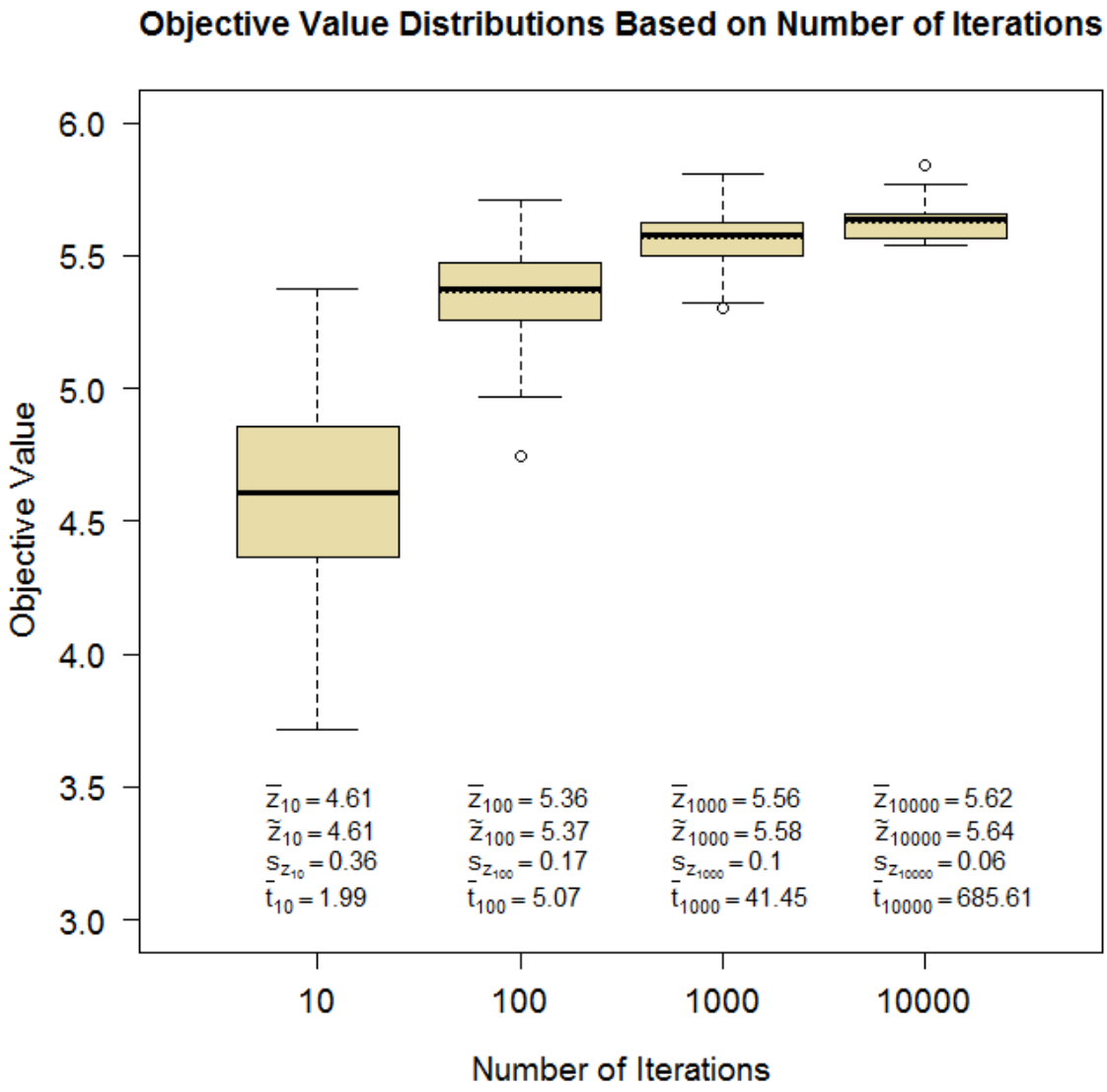

Figure 4.6: Box plots of objective values for cases 4-7.

	Mission Assignments			
Platform	Case 4	Case 5	Case 6	Case 7
1	1,6	1,6	1,6,14	1,6,14
2	2,6,13	4,6,9,16	3,6,13	1,6,13
3	10	10	1,10	5,10
4	1,6,16	2,7,13	4,11,16	4,11,16
5	2,12	1,12	2,11	2,9
6	3,9,14	1,13	5,9	5,9
7	4,7,9,15	2,6,11,15	2,6,9,15	2,6,10,15
8	6,12	1,12	4,7,12	4,7,12
9	4,8,15	3,8,14	5,8,15	3,8,15
OBJ	**4.953**	**5.059**	**5.369**	**5.658**

Table 4.13: Example mission assignments and objective values for cases 4-7.

Case 4 Example Output

Using a computational budget of 10 iterations per decision, the MCTS scheduling algorithm produced the schedule shown in table 4.14. Figure 4.7 gives a visual depiction of the mission coverage resulting from this schedule. The black horizontal lines indicate a mission that is covered, while the red horizontal lines indicate a mission that is left uncovered. It is important to note that these plots do not depict or otherwise account for how effective the platform is that is assigned to each mission. These mission effectiveness coefficients, given in table 4.12, are an important consideration in the assignment process. The runtime for this replication was 2.434 seconds.

Platform	Mission #	Take-off	Arrive	Depart	Land
4	1	-0.08	0.58	1.74	2.41
5	2	0.11	0.68	1.09	1.66
6	3	0.41	0.86	1.49	1.94
9	4	1.24	1.45	2.68	2.89
2	2	1.02	1.45	2.23	2.65
1	1	1.23	1.74	2.76	3.27
7	4	2.50	2.68	2.78	2.96
4	6	8.91	9.67	10.64	11.40
8	6	9.95	10.64	10.80	11.49
2	6	10.33	10.80	11.78	12.26
7	7	11.20	11.29	11.85	11.93
9	8	11.41	11.51	12.53	12.63
1	6	11.19	11.78	12.62	13.21
6	9	11.76	12.37	12.69	13.29
7	9	14.06	14.58	15.54	16.06
3	10	12.73	13.74	14.56	15.57
5	12	14.50	14.89	15.67	16.05
8	12	15.32	15.73	16.45	16.86
6	14	16.89	17.60	17.72	18.43
2	13	16.98	17.60	17.86	18.48
9	15	18.45	18.98	19.57	20.10
7	15	19.09	19.57	19.84	20.32
4	16	19.88	20.24	22.00	22.36

Table 4.14: Schedule produced for case 4.

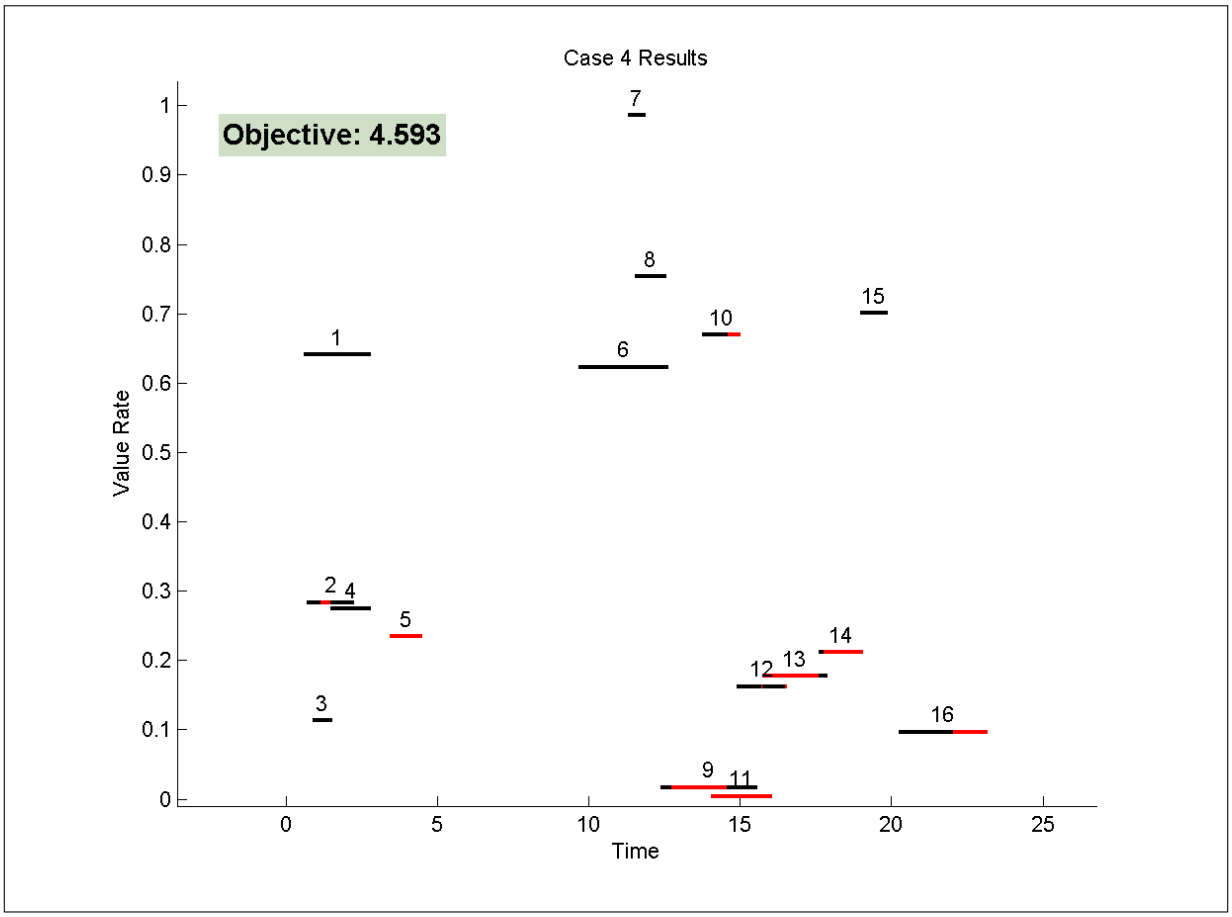

Figure 4.7: Visual depiction of schedule for case 4.

Case 5 Example Output

Using a computational budget of 100 iterations per decision, the MCTS scheduling algorithm produced the schedule shown in table 4.15. Figure 4.8 gives a visual depiction of this schedule. The runtime for this replication was 3.104 seconds.

Platform	Mission #	Take-off	Arrive	Depart	Land
1	1	0.07	0.58	1.73	2.25
7	2	0.27	0.68	1.88	2.30
9	3	0.43	0.86	1.49	1.92
2	4	1.26	1.45	2.78	2.97
6	1	1.26	1.73	2.32	2.79
4	2	1.21	1.88	2.23	2.90
5	1	1.76	2.32	2.75	3.31
8	1	2.15	2.75	2.76	3.36
2	6	9.19	9.67	10.65	11.13
7	6	10.18	10.65	11.75	12.21
4	7	11.15	11.29	11.85	11.98
9	8	11.41	11.51	12.53	12.63
1	6	11.15	11.75	12.62	13.21
2	9	14.88	15.41	15.54	16.07
3	10	12.73	13.74	14.56	15.57
7	11	14.34	14.87	15.84	16.37
8	12	14.47	14.89	15.61	16.02
6	13	15.02	15.73	15.84	16.55
5	12	15.35	15.73	16.51	16.90
4	13	16.40	17.39	17.86	18.85
9	14	17.09	17.77	18.06	18.74
7	15	18.51	18.98	19.84	20.32
2	16	20.01	20.24	21.72	21.95

Table 4.15: Schedule produced for case 5.

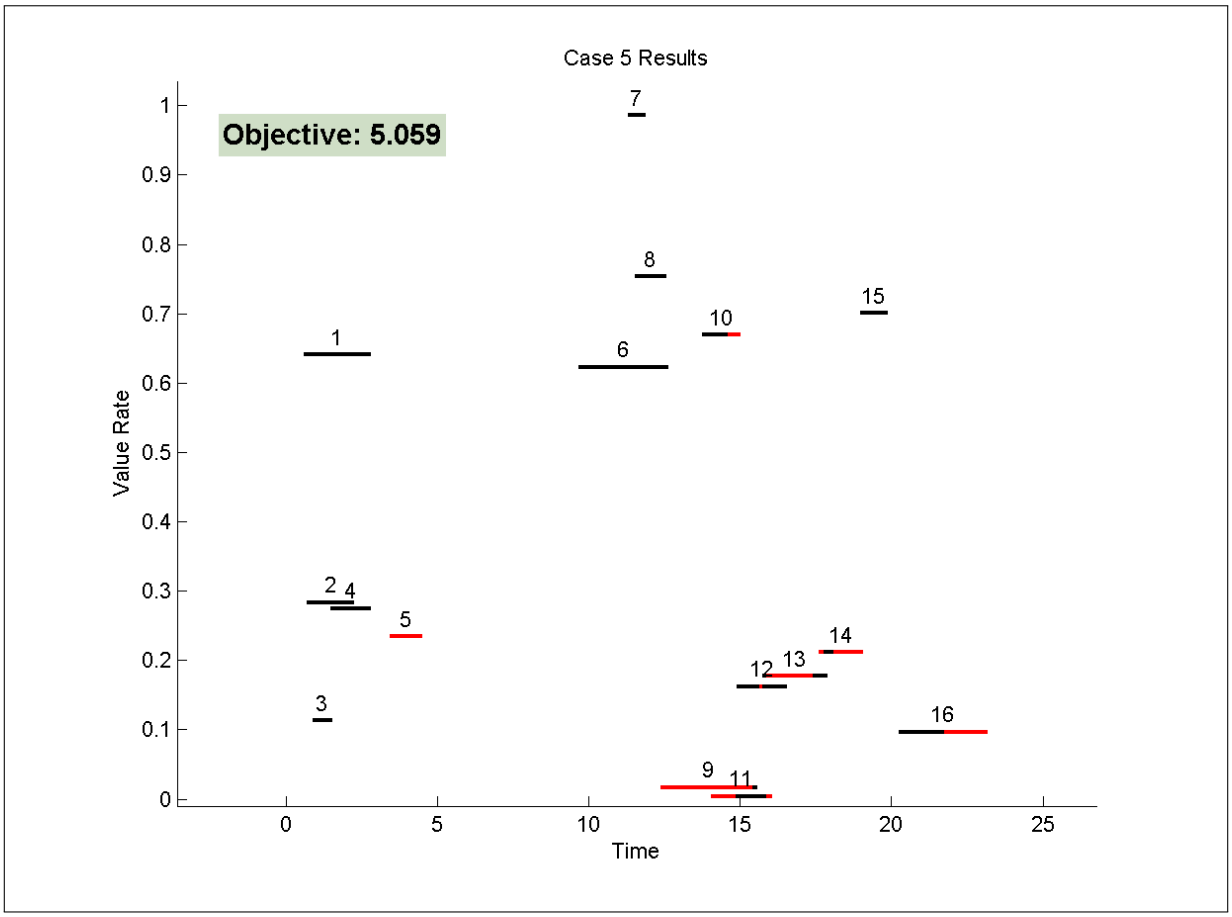

Figure 4.8: Visual depiction of schedule for case 5.

Case 6 Example Output

Using a computational budget of 1000 iterations per decision, the MCTS scheduling algorithm produced the schedule shown in table 4.16. Figure 4.9 gives a visual depiction of this schedule. The runtime for this replication was 25.21 seconds.

Platform	Mission #	Take-off	Arrive	Depart	Land
1	1	0.07	0.58	1.73	2.25
5	2	0.11	0.68	1.09	1.66
2	3	0.46	0.86	1.49	1.89
7	2	0.68	1.09	2.23	2.64
8	4	1.18	1.45	2.45	2.72
3	1	1.13	1.73	2.76	3.36
4	4	2.15	2.45	2.78	3.08
9	5	3.09	3.42	4.41	4.74
6	5	4.07	4.41	4.48	4.82
7	6	9.21	9.67	10.77	11.23
1	6	10.18	10.77	11.77	12.36
8	7	11.16	11.29	11.85	11.97
9	8	11.41	11.51	12.53	12.63
2	6	11.29	11.77	12.62	13.09
7	9	13.36	13.88	14.87	15.39
3	10	12.73	13.74	14.56	15.57
4	11	13.17	14.03	14.79	15.65
5	11	14.06	14.79	14.88	15.61
6	9	14.27	14.87	15.20	15.80
8	12	15.52	15.93	16.51	16.93
2	13	16.85	17.47	17.86	18.48
1	14	17.94	18.71	19.04	19.81
9	15	18.45	18.98	19.57	20.10
7	15	19.09	19.57	19.84	20.32
4	16	20.07	20.43	22.19	22.55

Table 4.16: Schedule produced for case 6.

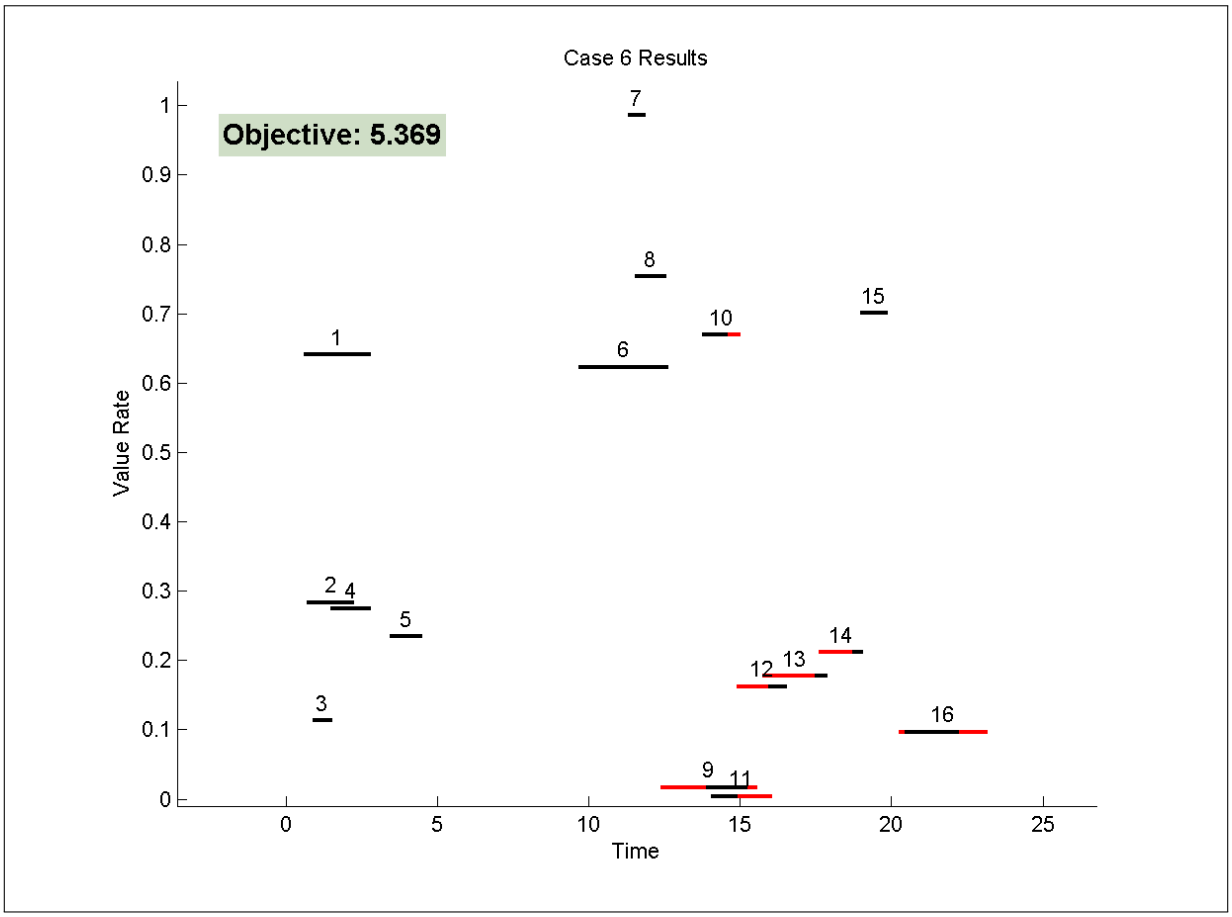

Figure 4.9: Visual depiction of schedule for case 6.

Case 7 Example Output

Using a computational budget of 10000 iterations per decision, the MCTS scheduling algorithm produced the schedule shown in table 4.17. Figure 4.10 gives a visual depiction of this schedule. The runtime for this replication was 411.49 seconds.

Platform	Mission #	Take-off	Arrive	Depart	Land
1	1	0.07	0.58	1.73	2.25
7	2	0.27	0.68	1.88	2.30
9	3	0.43	0.86	1.49	1.92
8	4	1.18	1.45	2.45	2.72
2	1	1.31	1.73	2.76	3.17
5	2	1.31	1.88	2.23	2.80
4	4	2.15	2.45	2.78	3.08
6	5	3.08	3.42	4.27	4.61
3	5	3.83	4.27	4.48	4.91
7	6	9.21	9.67	10.77	11.23
1	6	10.18	10.77	11.77	12.36
8	7	11.16	11.29	11.85	11.97
9	8	11.41	11.51	12.53	12.63
2	6	11.29	11.77	12.62	13.09
3	10	12.73	13.74	14.56	15.57
6	9	13.14	13.74	14.07	14.67
4	11	13.17	14.03	14.79	15.65
7	10	13.88	14.56	14.99	15.66
5	9	13.84	14.56	14.68	15.40
8	12	15.52	15.93	16.51	16.93
2	13	16.85	17.47	17.86	18.48
1	14	17.94	18.71	19.04	19.81
9	15	18.45	18.98	19.57	20.10
7	15	19.09	19.57	19.84	20.32
4	16	20.07	20.43	22.19	22.55

Table 4.17: Schedule produced for case 7.

4.4. ASC-U Implementation

The implementation of MCTS in the original ASC-U tool is ongoing. The results and documentation of this implementation will follow this report in a technical memorandum.

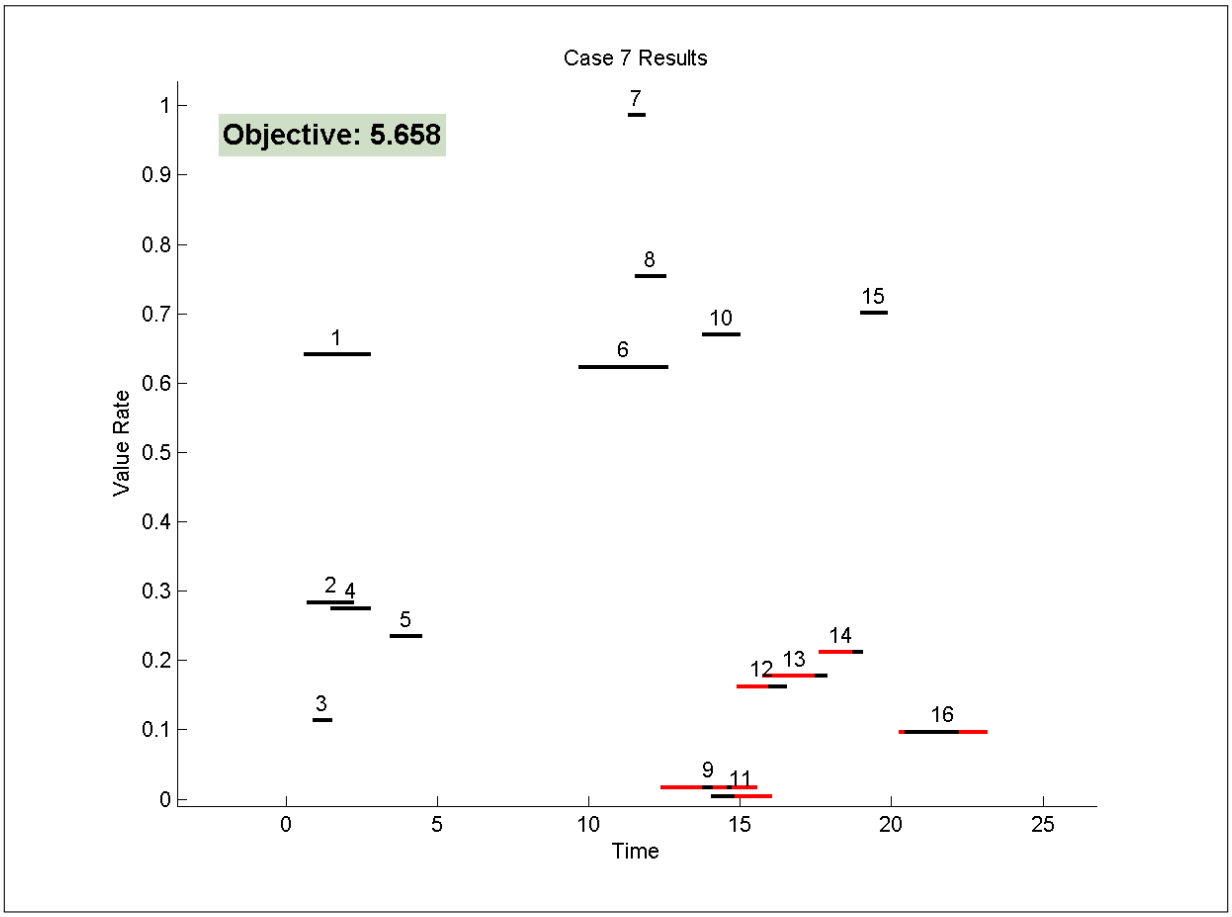

Figure 4.10: Visual depiction of schedule for case 7.

5. Future Implementation Directions

In this chapter we develop MCTS applications for future implementation in JDAFS and COMBATXXI.

5.1. JDAFS Implementation

In JDAFS we consider a scenario loosely based on one of the scenarios used in the Deployed Force Protection (DFP) study[10]. A coalition (blue) light infantry company consisting of three platoons and three 60-mm mortars (see figure 5.1) has established a combat outpost (COP) in order to conduct support and stability operations. One platoon secures the COP, one platoon mans an observation post and patrols the surrounding area, one platoon remains in the COP as a quick reaction force, and the mortars remain in the COP prepared to execute fire missions. The coalition forces have information that there are three enemy platoons operating in the area.

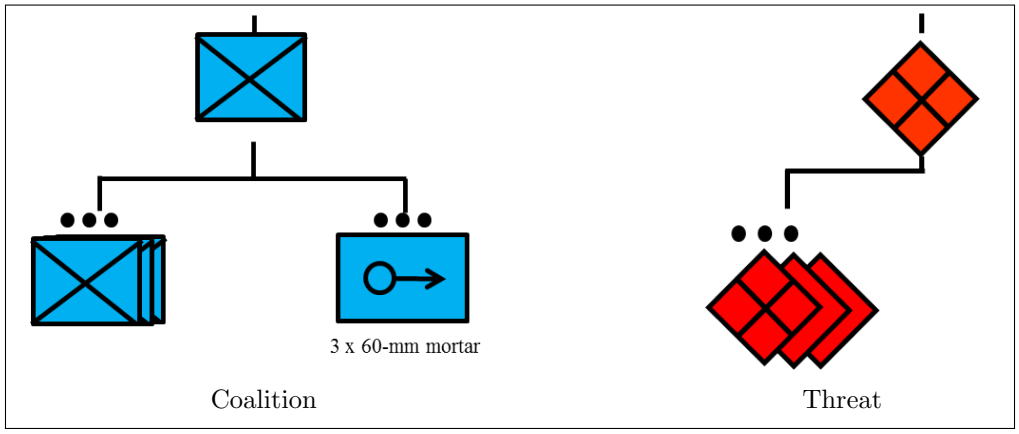

Figure 5.1: The coalition force and threat organizations for the JDAFS scenario.

The threat forces in this scenario consist of a company of dismounted infantry organized into three platoons and equipped with small arms (see figure 5.1). These platoons will move to the COP from the south and southwest. The threat will initiate the attack with one platoon that will maneuver to attack the COP from the southeast. Once the coalition forces have reacted to this initial attack, the enemy will attack the COP with its main effort, the remaining two platoons, from the southwest. Figure 5.2, adopted from [10], depicts this scenario. While this figure depicts many terrain features which were included in the COMBATXXI scenario build to support the DFP project, JDAFS does not model the effects of terrain so these features will not have any impact our implementation.

We employ MCTS in this scenario in JDAFS to determine which courses of actions produce the best results from the perspective of the coalition.

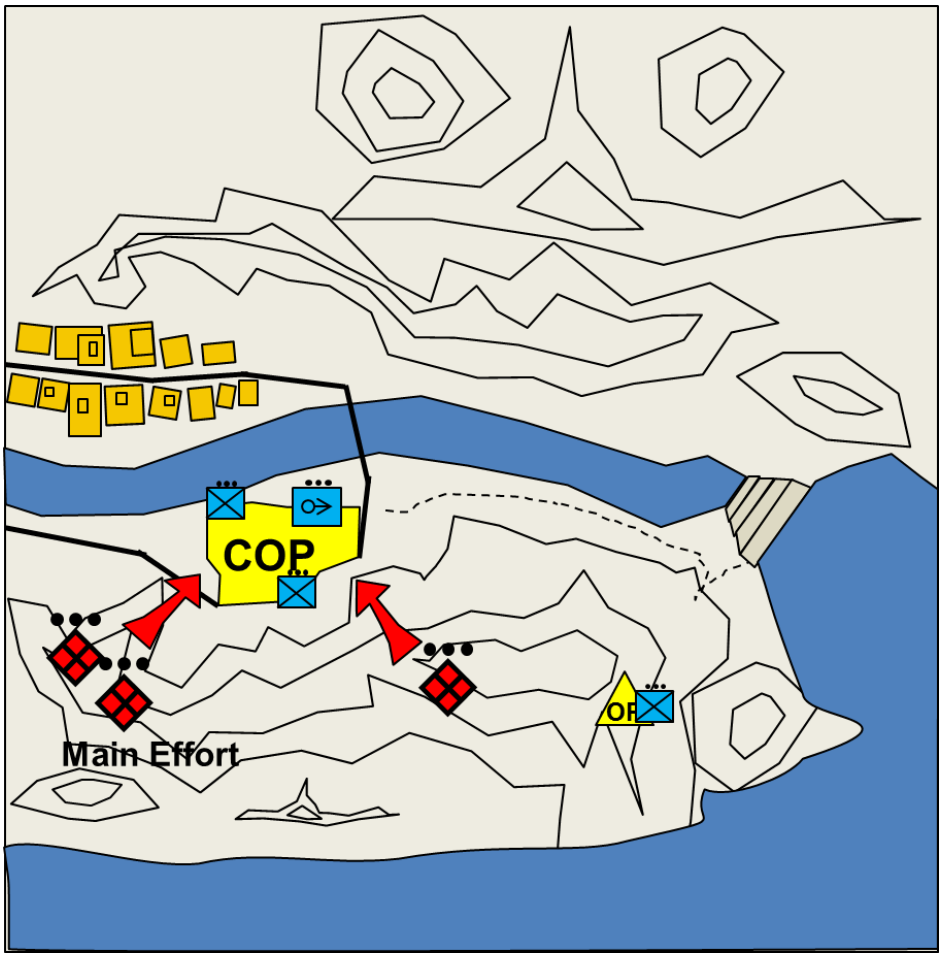

Figure 5.2: Concept for the JDAFS scenario.

5.1.1. Action Space

We assume there is a single decision maker controlling the actions of the blue elements. In defining the action space, we consider the set of information a decision-maker might use to generate courses of action. Let information vector

$$\mathbf{I} =< \mathbf{E}, \mathbf{F} >,$$

where $\mathbf{E} = \{E_1, E_2, \ldots, E_n\}$, the set of enemy entities detected by friendly forces, and $\mathbf{F} = \{F_1, F_2, \ldots, F_m\}$, the set of friendly (subordinate) elements available to task. We consider each friendly element and each detected enemy element as a single entity in this implementation.

We use the information vector, $\mathbf{I}$, to build the action space. We assume that, at each decision point, each friendly element can either attack one detected enemy or remain idle. An example decision might be

$$u = \begin{bmatrix} f_1 \rightarrow e_3 \\ f_2 \rightarrow e_2 \\ f_3 \rightarrow \text{Idle} \end{bmatrix},$$

which implies that friendly element "1" attacks detected enemy "3," friendly element "2" attacks detected enemy "2," and friendly element "3" remains idle. Note that there are $(n+1)^m$ possible decisions at each decision point in this action space, where n is the number of detected enemy elements and m is the number of friendly elements. Finally, we will not alter the JDAFS default behaviors, e.g., if an element comes under attack will automatically defend itself against the threat, regardless of current task.

In order to keep this application tractable, we examine only two decision points of interest in this scenario corresponding to the first and second enemy platoon detections. We will employ the JDAFS default behaviors, including assignments made by the constrained value optimization, to carry out the rest of the simulation.

5.1.2. Reward Function

We employ a normalized function of force difference to measure the utility of a course of action:

$$\Delta(j) = \frac{3 + x_{s(j)} - y_{s(j)}}{7},$$

where $x_{s(j)}$ is the number of blue forces alive in state $s(j)$ and $y_{s(j)}$ is the number of red forces alive in state $s(j)$.

5.1.3. Search Algorithm

We use the kSPO-MCTS algorithm described in paragraph 3.1.1. We set $k = 2$ samples per chance node. Considering the $a = (n+1)^m$ actions available at each decision point, the size of the fully-enumerated search tree for the first decision point is

$$k\left[\prod_{n=1}^{2} k\,(n+1)^m\right] = 10368 \text{ nodes.}$$

In the first iteration, according to the kSPO MCTS algorithm, the program will generate two complete simulation states based on the partial information available to the blue forces, including the knowledge that there are a total of three threat platoons in the area. The MCTS algorithm proceeds from these two fully defined states.

5.1.4. Default Algorithm

The default algorithm for this implementation will be to choose a decision from the action space at random.

5.1.5. Status of Implementation

The project team has laid some of the programming groundwork for implementation into JDAFS. In particular, we have created a MCTS method to replace the constrained value optimizer (CVO) that makes assignments. We have also created the initial test scenario, discussed in this chapter.

Figure 5.3: COMBATXXI MCTS test scenario.

5.1.6. Potential Insights

This implementation can provide insights on the applicability of MCTS as an analysis tool in a recursive simulation environment in several roles, including:

- Mission command analysis.
- Analysis of optimal decisions and decision making.
- Determining the influence of capabilities on decisions and outcomes.
- Sequential design of experiments.

5.2. CombatXXI Implementation

In this section we develop a proof of principle implementation of MCTS for use in COMBATXXI. COMBATXXI is a high resolution simulation environment, making computational time and capacity more important to manage in any recursive simulation efforts. Our scenario consists of a group of "red" riflemen are advancing on a single "blue" grenadier in a fixed position (see figure 5.3). Armed with an assault rifle (60 rounds) and a grenade launcher (12 grenades), the grenadier must decide with which weapon to engage the enemy. By varying the number of red troops advancing on the blue grenadier, we can use the results of MCTS runs to gain insight into what situations the grenadier should choose each weapon over the other in order to get the best effects, and when it might be best to switch weapons during engagements. The red elements will execute COMBATXXI default behaviors, i.e., they will engage the blue entity once they it is within range. For the purpose of this analysis, the advancing red troops will remain relatively concentrated. The results of this analysis might be useful in informing the CombatXXI weapons selection behavior, which currently simply selects weapons from a weapon-priority list.

5.2.1. Action Space

We suggest an Information Set UCT (ISUCT)[2] MCTS algorithm for this scenario. We use the information available to the blue grenadier to populate the search tree. This information consists of:

- How many red troops the grenadier can see.
- How much ammunition is remaining for each weapon system.

By abstracting the MCTS state space to this information, we limit the size and dimensionality of the search tree.

The blue grenadier will only make decisions using MCTS if all of the following criteria apply (these criteria define the decision points in the simulation):

- The grenadier can see one or more enemy elements.
- The grenadier still has the capability (i.e., the ammunition) to use both weapons systems.
- The grenadier is not currently engaging the enemy based on a previous decision.

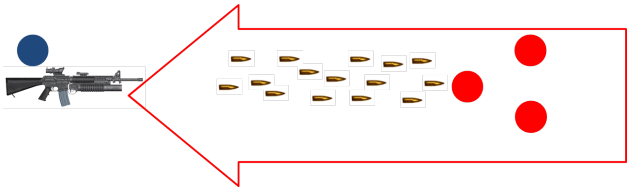

Figure 5.4: Decision to engage with the rifle.

At each decision point the blue grenadier takes one of the following two actions (i.e., these actions comprise the action space):

- Engage with fifteen rounds from the assault rifle (see figure 5.4), or
- Engage with three grenades from the grenade launcher (see figure 5.5).

5.2.2. Reward Function

The reward function at the end of each run is:

$$\Delta(j) = 1 - \frac{y_{s(j)}}{y_{s_0}},$$

where $y_{s(j)}$ is the number of red troops alive at the end of the simulation run and y_{s_0} is the number of red troops alive at the beginning of the simulation run.

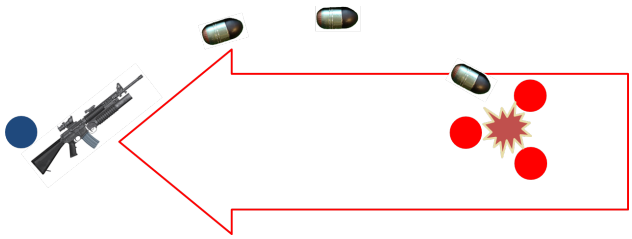

Figure 5.5: Decision to engage with grenades.

5.2.3. Search Algorithm

The UCT algorithm serves as the search algorithm for this implementation. We allow for stochastic outcomes by maintaining the chance node structure developed in the kSPO algorithm, but we do not control the sampling from stochastic outcomes. Rather, we let the outcomes develop stochastically but keep in mind that a stochastic outcome only constitutes a unique state (i.e., node) if the decision information (number of red elements visible to blue and ammunition remaining) are different from all other outcomes.

5.2.4. Default Algorithm

We consider three default algorithms for this implementation:

- Random weapon selection at decision points.
- COMBATXXI default behavior (i.e., select weapons based on a priority list).
- The fires allocation algorithm, discussed in section 3.3.

5.2.5. Status of Implementation

Currently we have not expended any computation effort into the COMBATXXI implementation. We plan to focus on this implementation in future work.

5.2.6. Potential Insights

This implementation can provide insights on the applicability of MCTS as an analysis tool in a high-resolution simulation environment, including:

- Scalability.
- Analysis of optimal decisions and decision making.
- Insight into the impacts of default behaviors.
- Determining the influence of capabilities on decisions and outcomes.

6. Summary and Conclusions

6.1. Summary

In this project we developed implementations of the Monte Carlo Tree Search algorithm for applications in military simulation environments and military analyses:

- We developed the kSPO and the ISPO algorithms for implementation in stochastic, partially observable games and simulations. We produced a simple prototype for the kSPO and observed the results, which demonstrated a proof-of-principle application. The Javascript source code for this implementation is available from the authors upon request. During this implementation we developed workarounds for the requirement to reproduce state space in a stochastic, partially observable simulation environment. These workarounds include
  - Running the simulation from the start point with the same input data and random seed to arrive at the same point.
  - Abstracting state space from the simulation space for the purpose of building a simplified game tree from a more complicated simulation state space.

  We also determined that limits on computational resources would naturally limit any application of MCTS in military simulation. It is simply infeasible to have a large number of entities all simultaneously running independent MCTS algorithms to make decisions. MCTS will be useful if applied on a small, finite action space of interest, identified in advance. Determining which decisions go into this action space involves deciding which decisions involve the "tripping point" issues that serve to inform the analysis.
- We implemented MCTS as a method for producing good, feasible schedules in a UAV assignment and scheduling problem. The results of this implementation demonstrated the usefulness of MCTS methods in other types of TRAC analyses, especially those involving platform assignment, scheduling, or allocation.
- We outlined scenarios and implementations for MCTS methods into JDAFS, as a tool for mission command analysis, and into COMBATXXI, as a tool for selecting the best weapon to use during an engagement.

In addition to the work with MCTS, the project team developed a fires allocation algorithm that inputs kill probabilities from multiple entities, each with one or more weapon systems, assigned to two opposing forces and computes the optimal fires assignment for each entity. We have produced a python-coded behavior for implementing this algorithm into COMBATXXI.

6.2. Conclusions

From our work on this project, we arrive at the following conclusions.

1. From the initial prototype implementation, we found that MCTS can function as a tool for autonomous decision-making in military simulations, or for analyzing the impact of decision-making in simulation environments. However, any application must consider the computational resources required. The next step in this line of research is to develop a working implementation in COMBATXXI or JDAFS for further analysis.
2. From the ASCU implementation, we identified that MCTS and similar AI methods might have applications beyond mission command and decision analyses in military simulation environments. In addition to completing an implementation in the original ASC-U environment, the next step in this line of research is to use MCTS to solve problems currently assigned to the Joint Platform Allocation Tool (JPAT), and compare performance and speed.
3. Finally, the fires allocation algorithm provides an efficient way of determining fires assignments in a two-sided engagement involving more than two entities. The next step in this line of research is to implement this algorithm as an entity weapon selection behavior, or as a fire planning algorithm to represent fire direction center behavior in COMBATXXI. We also intend to use this behavior in conjunction with the MCTS implementations in simulation environments, as outlined in section 3.3.

MCTS methods provide a means for arriving at a good decision or sequence of decisions from a complicated action space in a short amount of time. We look forward to developing more applications for MCTS to inform TRAC analyses in the future.

Appendix A. Fires Allocation Algorithm in MATLAB

In this appendix we provide the functional code that executes the fires allocation algorithm and an example MATLAB script file that provides the inputs, calls the function, and saves the output fires allocation to memory.

A.1. Algorithm Code

The following code implements the fires allocation algorithm, presented in section 3.3, in MATLAB^{TM}. This code takes as its input matrices representing all "blue" and "red" elements' weapon systems and kill probabilities, for all weapon-shooter-target pairings, and outputs the optimal fire allocation.

solveFireAllocationMP.m

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% In fire allocation game, there are two players A (or Blue) and B (or Red)
% This program computes the probability that Blue wipes out Red before Red
% wipes out Blue It also returns the optimal fire allocation for Blue on
% Red, and for Red on Blue
%
% Blue has m units and Red has n units.  Each unit has up to r different
% ''choices of fire allocation'' A fire allocation specifies a weapon to use
% and the target, therefore a vector of kill rates on all enemy targets
% lambda is a 3 dimensional matrix size of (r, n, m), with lambda(k,j,i)
% being Blue unit i's kill rate at Red unit j, if Blue i uses the k-th
% choice theta is a 3 dimensioanl matrix size of (r, m, n), with
% theta(k,i,j) being Red unit j's kill rate at Blue unit i, if Red j uses
% the k-th choice
%
% There are three return values V: probability A wins x:  Blue's optimal
% choice of fire allocation, m by 1 matrix y:  Red's optimal choice of fire
% allocation, 1 by n matrix
%
% In order to compute V, the program actually computes the optimal
% allocation for each subset of Blue vs each subset of Red, recursively,
% starting with the smallest subset.  However, the program only outputs the
% value at the beginning of the game
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

function [V x y] = solveFireAllocationMP(lambda, theta)
    [r n m] = size(lambda); %Blue has m units; Red has n units; each unit has l kill rate profile
    mm=2m-1; %number of states for Blue; for instance, state 6 -> [0 1 1] -> Blue unit 1 is dead; 2 and 3
alive
    nn=2n-1; %number of states for Red
    % the number of states is mm * nn

    lambda = max(lambda, 10^(-12) * ones(r,n,m)); % use 10^(-12) to replace a kill rate of 0
    theta = max(theta, 10^(-12) * ones(r,m,n)); % use 10^(-12) to replace a kill rate of 0

    Vmatrix = - ones(mm,nn); %Vmatrix is the matrix for game value in each state, -1 indicates the value has
not been computed

    numUnitMatrix = zeros(mm,nn); %compute the number of alive units in each state
    for i=1:mm
        for j=1:nn
```

```
            numUnitMatrix(i,j) = sum(de2bi(i,m)) + sum(de2bi(j,n));
        end
    end

    % find V (Blue's win prob) for all cases when each team has 1 unit left
    for i=1:mm
        for j=1:nn
            if numUnitMatrix(i,j) == 2 % check if each team has 1 unit
                [value blueAlive] = max(de2bi(i,m));
                [value redAlive] = max(de2bi(j,n));
                blueRate = max(lambda(:, redAlive, blueAlive));
                redRate = max(theta(:, blueAlive, redAlive));
                Vmatrix(i,j) = blueRate / (blueRate + redRate);
            end
        end
    end

    for totalNumUnit = 3:(m+n) % iteratively compute V on the number of total units, starting from 3
        for i=1:mm
            for j=1:nn
                if numUnitMatrix(i,j) == totalNumUnit

                    blueState = de2bi(i,m)'; % a column of 1s and 0s
                    redState = de2bi(j,n); % a row of 1s and 0s

                    % remove each Blue unit and find the resulting win
                    % probability in addition, the largest would be the
                    % best lower bound for V
                    vLB = 0; % 0 is by default a lower bound for win prob
                    vBlueOneDown = 2 * ones(m,1);   %for Blue win probabiliy if Blue loses unit i
                    % if Blue does not have unit i, then the entry has
                    % value 2.  This way, none of Red unit will fire at Blue
                    % unit i

                    if sum(blueState) == 1 % if Blue has only 1 unit, vLB = 0 no change
                        vBlueOneDown(find(blueState==1)) = 0; % vBlueOneDown has one 0 and all others -1
                    else
                        %if Blue has 2 or more units, we can get a better
                        %lower bound by removing one Blue unit at a time
                        for k=1:m
                            if blueState(k)==1 % if Blue unit k is still alive, remove it to see the resulting
win probability
                                vBlueOneDown(k) = Vmatrix(i-2^(k-1), j);
                                vLB = max(vLB, Vmatrix(i-2^(k-1), j));
                            end
                        end
                    end

                    % remove each Red unit and find the resulting win
                    % probability in addition, the smallest would be the
                    % best upper bound for V
                    vUB = 1; % 1 is by default an upper bound for win prob
                    vRedOneDown = - ones(1,n);   %for Blue win probabiliy if Red loses unit j
                    % If Red does not have unit j, then the entry has value
                    % -1.  This way, none of Blue unit will fire at Red unit
                    % j
                    if sum(redState) == 1 % if Red has only 1 unit, 1 is the best upper bound
                        vRedOneDown(find(redState==1)) = 1; % vRedOneDown has one 1 and all others -1
                    else
                        %if Red has 2 or more units, we can get a better
                        %upper bound by removing one Red unit at a time
                        for k=1:n
                            if redState(k)==1 % if Red unit k is still alive, remove it to see the resulting
win probability
                                vRedOneDown(k) = Vmatrix(i, j-2^(k-1));
                                vUB = min(vUB, Vmatrix(i, j-2^(k-1)));
                            end
```

```
                    end
                end

                % begin iteration
                V = (vLB+vUB)/2; % intiail guess of V

                % initialize x, xOld, and y, yOld:  fire allocation
                % (which kill rate profile to use)
                x = ones(m,1); % column of length m; Blue unit i uses profile x(i)
                xOld = zeros(m,1); % keep track of x from the previous iteration
                y = ones(1,n); % row of length n; Red unit j uses profile y(j)
                yOld = zeros(1,n); % keep track of y from the previous iteration

                % iteratively compute the fire allocation; stop only if
                % the fire allocations are the same as the previous
                % iteration
                while (isequal(x,xOld) + isequal(y,yOld) < 2) % stop if x==xOld and y==yOld
                    xOld = x;
                    yOld = y;
                    delta = (vRedOneDown - V) .* redState; % a row of length n
                    x = blueState .* optAllocate(lambda, delta); %optimal policy for Blue
                    % if Blue does not have unit i, then the ith place
                    % is 0
                    delta = (V - vBlueOneDown) .* blueState; % a column of length m
                    y = redState .* optAllocate(theta, delta')'; %optimal policy for Red (transpose)
                    % if Red does not have unit j, then the jth place
                    % is 0
                    alpha = zeros(1,n); % Blue (total) rate at killing Red unit j, j=1,...,n
                    for k=1:m
                        if blueState(k) == 1 % if Blue unit i still alive
                            alpha = alpha + lambda(x(k), :, k);
                        end
                    end
                    alpha = alpha .* redState; % kill rate is only "real" if the corresponding Red unit is
still alive
                    beta = zeros(m,1); % Red (total) rate at killing Blue unit i, i=1,...,m
                    for k=1:n
                        if redState(k) == 1 % if Red unit j still alive
                            beta = beta + theta(y(k), :, k)';
                        end
                    end
                    beta = beta .* blueState; % kill rate is only "real" if the corresponding Blue unit is
still alive
                    % new value for this iteration
                    V = (sum(alpha .* vRedOneDown) + sum(beta .* vBlueOneDown)) / (sum(alpha) +
sum(beta));
                end

                Vmatrix(i,j) = V;        %end of iteration
                % at this point, we have computed on entry in the
                % matrix Vmatrix; need to repeat this procedure many
                % times
            end
        end
    end
end

% At the end of iteration, we just computed the optimal policy when all
% units are alive V, x, y correspond to the value at the beginning, when
% both A and B have full force, therefore the output

% For 3-dimensional kill rate matrix, and the ``value'' of killing each
% opposing unit, compute the optimal fire allocation
%
% lambda is the kill rate matrix (size r x n x m); delta a row (size 1 x n)
% optAlc is a column (m x 1), optimal fire allocation
function optAlc = optAllocate(lambda, delta)
```

```
delta;
[r n m] = size(lambda);
productMat = zeros(m, r); % initialize the product
for i=1:m
    productMat(i,:)  = (lambda(:,:,i) * delta')';
end
[maxValue optAlc] = max(productMat, [], 2); % in case of tie, use the profile with smallest ID
```

A.2. Example Use Script

The following MATLAB script provides example inputs for the fires allocation algorithm, calls the fires allocation function (code given in section A.2 above), and returns the optimal fires allocation.

example.m

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% An example for the fire allocation game, when each unit has multiple
% weapon platforms
%
% Suppose that A (or Blue) has m units and B (or Red) has n units
% Each unit has up to r different "choices of fire allocation"
%
% A fire allocation specifies a "weapon" to use and the "aim", which
% induces a vector of kill rates on all enemy targets
%
% lambda is a 3 dimensional matrix size of (r, n, m), with lambda(k,j,i)
% being Blue unit i's kill rate on Red unit j, if Blue i uses the k-th
% choice of fire
%
% theta is a 3 dimensioanl matrix size of (r, m, n), with theta(k,i,j)
% being Red unit j's kill rate on Blue unit i, if Red j uses the k-th
% choice of fire
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Here is an example when Blue has m=2 units and Red has n=3 units; each unit
% has up to r=4 choices of fire allocation

m = 2;
n = 3;
r = 4;

lambda = zeros(r, n, m);
theta = zeros(r, m, n);

lambda(:,:,1) = [
1.3 0 0; % M16 aiming at Red unit 1
0 1.6 0; % M16 aiming at Red unit 2
0 0 0.7; % M16 aiming at Red unit 3
0.6 0.4 0.6]; % M203 aiming at the center of 3 Red units
lambda(:,:,2) = [
1.5 0 0;
0 1.8 0;
0 0 0.4;
0.3 0.3 0.2];
```

```
theta(:,:,1) = [
0.7 0; % M16 aiming at Blue unit 1
0 1.3; % M16 aiming at Blue unit 2
0.1 0.3; % M203 aiming at Blue unit 1 (collateral damage possible)
0.3 0.1]; % M203 aiming at Blue unit 2 (collateral demage possible)
theta(:,:,2) = [
0.6 0;
0 0.7;
0.4 0.4;
0 0]; % use a vector of 0 to deactive this choice
theta(:,:,3) = [
1.5 0;
0 1.1;
0.3 0.4;
0 0];

% use this function to compute
% V: the probability A will win
% x:  A's optimal choice of fire allocation
% y:  B's optimal choice of fire allocation

[V x y] = solveFireAllocationMP(lambda, theta)
```

This page intentionally left blank.

Appendix B. Prototype MCTS ASCU Implementation in MATLAB

This appendix provides documentation for the MCTS ASCU prototype implementation in MATLAB™. This code consists of several basic functions that contribute to the execution of the MCTS algorithm:

- `ASCUMCTS.m`
- `ASCUdefault.m`
- `createchilds.m`
- `MDupdate.m`

In addition, the code includes several additional functions that assist in analyzing and documenting the results:

- `missionplot.m`
- `allassignmentplot.m`
- `assignmentplot.m`
- `singleassignmentplot.m`
- `document.m`

Finally, the MATLAB files include a script file, `ASCU.m`, that reads the input database files, executes the MCTS algorithm, and saves and analyzes the results.

We now provide brief documentation for each of these functions and scripts, along with a printed copy of the code itself.

B.1. `ASCU.m` Script File

The `ASCU.m` MATLAB script file serves as the control for executing the MCTS algorithm. The script begins by clearing the memory and then reading in the three input data arrays. The arrays currently come from three sheets in a single Microsoft™ Excel file.

- `MDin`. This array contains the mission data, listed in order by increasing start times. It comes from the sheet labeled 'MD' in the input Excel file, and consists of six columns:

1. A unique number identifying the mission.
2. The mission start time.
3. The mission duration.
4. The x coordinate of the mission location.
5. The y coordinate of the mission location.
6. The value rate for the mission.

- `platform`. This array contains the platform data. It comes from the sheet labeled 'platform' in the input Excel file, and consists of three columns:

  1. The platform endurance.
  2. The platform speed.
  3. The platform reset time.

- `perform`. This array contains the platform performance data. It has one row for every platform and one column for every mission. Entry (i, j) in this array is the performance coefficient for platform i (the ith row in the `platform` array) against mission j (the jth row in the `mission` array).

After reading the inputs, the `ASCU.m` script initializes some additional parameters, including the values for `const`, the constant C_p in the UCT expression, and `cmax`, the number of MCTS iterations allocated to each decision (i.e., the computational budget). The state platform availability vector `savail` is set to $-\infty$ for each platform, indicating that they have not performed any previous missions and area all ready to take off to fulfill mission requirements. The script also initializes an `assignments` array.

Once all the parameters have been initialized, the script loops through the current mission array. For each mission, the script executes the `ASCUMCTS.m` function. The script records the decision output by the `ASCUMCTS.m` function and updates the current mission array, current mission index (for assignment), and platform availability vector for the next loop.

This loop terminates once the current mission index exceeds the number of rows in the current mission array. At that point, each of the assignment decisions is stored in the `assignments` array. The script then computes the objective value for the assignment and computes and records the runtime. Following these steps, the script executes and saves the assignment plots. The first plot generated is either the `allassignmentplot.m` or the `assignmentplot.m` (see explanations below). Only one of these plots should be generated; the other should be commented out according to user preference. Following this plot, the script generates a `singleassignmentplot.m` for each platform.

Finally, the script computes the coverage percentage, in case that value is of interest to the user. The script runs the `document.m` function, which produces a LaTeXinput file containing tabulated input data and results for the test case. Finally, the script compiles the LaTeXfile into a portable document format (PDF) file for easy viewing by the user.

The code for the ASCU.m MATLAB script is below.

ASCU.m

```
%ASCU Run script
%Written by Chris Marks
%read inputs
%clear
%scenario = 3;
starttime = clock;
MDin = xlsread(sprintf('../case%u/input%u.xlsx',scenario,scenario),'MD'); %List missions in order.
platform = xlsread(sprintf('../case%u/input%u.xlsx',scenario,scenario),'platform');
perform = xlsread(sprintf('../case%u/input%u.xlsx',scenario,scenario),'perform');
%initialize
MD = MDin;
const = 1/sqrt(2)*(MDin(:,6)'*MDin(:,3));
%cmax = 100;
savail = zeros(size(platform,1),1);
savail(:,1) = -inf;
assignments = zeros(1,6); %platform, mission, takeoff, arrive, depart, land
counter = 1;
currentMD=1;
while(currentMD <= size(MD,1))
  [decision MD currentMD newstate] = ASCUMCTS(currentMD,MD,platform,savail,const,cmax,perform);
  assignments(counter,:)  = decision;
  counter = counter + 1;
  savail = newstate;
end
a = [assignments(assignments(:,1) =0,1) assignments(assignments(:,1) =0,2)];
clear performances;
for i = 1:size(a,1)
  performances(i) = perform(a(i,1),a(i,2));
end
objective = ((assignments(assignments(:,1) =0,5) - assignments(assignments(:,1) =0,4)) .* performances')' *
MDin(assignments(assignments(:,1) =0,2),6);
endtime = clock;
iterationtime = endtime-starttime;
totaltime = 60*60*iterationtime(4) + 60*iterationtime(5)+iterationtime(6);
h = figure('Color','w');
title(sprintf('Case %u Results',scenario));
maxx = max((MDin(:,2)+MDin(:,3))');
maxy = max(MDin(:,6)');
scale = [-(2.5/16)*maxx maxx*(1+2.5/16) -(1/100)*maxy maxy*(1+5/100)];
axis(scale);
xlabel('Time');
ylabel('Value Rate');
missionplot(MDin,scale);
%allassignmentplot(assignments,MDin);
assignmentplot(assignments,MDin);
text(scale(1)+0.05*(scale(2)-scale(1)),scale(4)-0.075*(scale(4)-scale(3)),['Objective:  '
num2str(objective,4)],'EdgeColor','g','BackgroundColor','g','FontSize',14,'FontWeight','bold');
saveas(h,sprintf('../case%u/results%u.png',scenario,scenario),'png');
close(h);
for p = 1:size(platform,1)
  h = figure('Color','w');
  title(sprintf('Case %u Platform %u Schedule',scenario,p));
  maxx = max((MDin(:,2)+MDin(:,3))');
  maxy = max(MDin(:,6)');
  scale = [-(2.5/16)*maxx maxx*(1+2.5/16) -(1/100)*maxy maxy*(1+5/100)];
  axis(scale);
  xlabel('Time');
  ylabel('Value Rate');
  missionplot(MDin,scale);
  singleassignmentplot(assignments,MDin,p);
  contr = ((assignments(assignments(:,1)==p,5) - assignments(assignments(:,1)==p,4)) .*
perform(p,assignments(assignments(:,1)==p,2))')' * MDin(assignments(assignments(:,1)==p,2),6);
```

```
  text(scale(1)+0.05*(scale(2)-scale(1)),scale(4)-0.075*(scale(4)-scale(3)),['Contribution:  '
num2str(contr,4)],'EdgeColor','g','BackgroundColor','g','FontSize',14,'FontWeight','bold');
  saveas(h,sprintf('../case%u/results%up%u.png',scenario,scenario,p),'png');
  close(h);
end
assign = assignments(assignments(:,1) =0,:);
document(scenario,assign);
coverage = (sum(assignments(:,5)-assignments(:,4)))/sum(MDin(:,3));
cd(sprintf('../case%u/',scenario));
dos('pdflatex readme.tex');
cd('../Current version/');
%Add multiple LRS capability.
%Add mobile LRS capability.
%Add GRS constraint on number of aircraft in the air
%Add capability to go directly from one mission to the next.
%Add FARP capability.
%Maximum mission value cut-off.
```

B.2. MCTS Execution Functions

We now provide documentation and code for the five functions that execute the MCTS algorithm.

B.2.1. `ASCUMCTS.m`

The `ASCUMCTS.m` function receives the following inputs:

- `currentMD`. The index of the current decision mission in the mission array (`MDin`).
- `MDin`. The mission array.
- `platform`. The platform array.
- `savail`. An array that tracks the most recent landing times of all of the platforms in the `platform` array.
- `const`. The constant C_p, in the UCT expression (see section 2.1.1).
- `cmax`. The computational budget, B, described in section 4.2.
- `perform`. The performance array.

The function returns the following outputs:

- `decision`. This vector includes a platform, a mission assignment, a take-off time, an on-station time, a mission area departure time, and a landing time.

- `MDout`. This vector contains an updated mission array for use as an input to follow-on decisions.
- `missionout`. This integer is the index of the next decision mission in the mission array for future missions, `MDout`.
- `newstate`. This array provides updated platform landing times, including those related to the `decision` returned.

The function initializes a `tree` array and a `state` array for storing the information structure necessary to support the MCTS algorithm. It executes the tree policy according to the UCT algorithm. It calls the function `MDupdate.m` in order to dynamically update the mission array, `MD`, based on the decision path the tree policy follows. Each time it comes to a previously unexplored node, it:

1. Executes the `createchilds.m` function.
2. Updates the `tree` and `state` parameters with the output.
3. Executes the `ASCUdefault.m` function.
4. Backpropagates the reward.
5. Steps the `iteration` counter.

Once the `iteration` counter reaches `cmax`, the function terminates the tree policy iterations, selects the best decision node from the set of root node children based on highest average reward, updates the mission array `MDout`, and returns the results.

The code for `ASCUMCTS.m` is below.

ASCUMCTS.m

```
function [decision MDout missionout newstate] = ASCUMCTS(currentMD,MDin,platform,savail,const,cmax,perform)
%function ASCUMCTS
%Written by Chris Marks
%Pre-process current mission demand.
totalreward = MDin(currentMD:end,3)'*MDin(currentMD:end,6);
%Create tree matrix and state array.
node = 1;
tree(node,:)  = [0 currentMD 0 0 0 0 MDin(currentMD,1) 0 0]; %parent, MD, platform assign, n , reward(back),
reward(forward) , mission number, arrive, depart
state(:,node) = savail; %last land times.
%Begin iterations
iteration = 0;
MD = MDin;
while(iteration < cmax)
  if tree(node,2) == (size(MD,1) + 1) % we have already reached a terminal state.
    r = tree(node, 6);
    while node  = 0;
      tree(node,4) = tree(node,4)+ 1;
      tree(node,5) = tree(node,5) + r;
```

```
      node = tree(node,1);
    end
    iteration = iteration + 1;
    node = 1;
    MD = MDin;
  else % we have not reached a terminal state.
    if tree(node,4)==0 %node has not been visited
      %create children
      [newnodes newstates] =
createchilds(MD(tree(node,2),:),platform,state(:,node),tree(node,:),node,tree(node,2),perform);
      tree(size(tree,1)+1:size(tree,1)+size(newnodes,1),:)=newnodes;
      state(:,size(state,2)+1:size(state,2)+size(newstates,2))=newstates;
      %Run the default algorithm
      r = ASCUdefault(tree(node,:),state(:,node),MD,platform,perform);
      %backpropagate
      while node  = 0;
        tree(node,4) = tree(node,4)+ 1;
        tree(node,5) = tree(node,5) + r;
        node = tree(node,1);
      end
      iteration = iteration + 1;
      node = 1;
      MD = MDin;
    else
      %select a child IAW the tree policy
      UCT = zeros(sum(tree(:,1)==node),2);
      ind = 1;
      for n=find(tree(:,1)==node)'
        if tree(n,4)==0
          UCT(ind,:)  = [n inf];
        else
          UCT(ind,:)  = [n, tree(n,5)/(totalreward*tree(n,4))+2*const*sqrt(2*log(tree(node,4))/tree(n,4))];
        end
        ind = ind + 1;
      end
      if sum(UCT(:,2)==max(UCT(:,2))) > 1 %There exist ties.
        ties = UCT(UCT(:,2)==max(UCT(:,2)),1);
        node = ties(randi(length(ties)));
      else
        node = UCT(UCT(:,2)==max(UCT(:,2)),1);
      end
      [MD tree(node,2)] = MDupdate(MD,tree(tree(node,1),2),tree(node,9));
    end
  end
end
r=0;
for n = find(tree(:,1)==1)'
  if (tree(n,5)/tree(n,4))>=r
    r = tree(n,5)/tree(n,4);
    if currentMD==size(MD,1)
      bests = 'NA';
    else
      bests = state(:,n);
    end
    bestn = tree(n,:); %node, parent, MD, platform assign, n , reward(back), reward(forward)
  end
end
%If r = 0, do what? Should automatically make the "no assignment" because
%that is the last child node created.
%Re-calculate timeon, timeoff, update mission matrix
if (bestn(3) == 0)
  decision = [0 bestn(7) 0 0 0 0]; %platform, mission, takeoff, arrive, depart, land
else
  dist = sqrt(MDin(currentMD,4)^2 + MDin(currentMD,5)^2); %distance to mission
  traveltime=dist/platform(bestn(3),2);
  decision = [bestn(3) bestn(7) bestn(8)-traveltime bestn(8) bestn(9) bestn(9)+traveltime]; %platform,
mission, takeoff, arrive, depart, land
end
```

```
newstate = bests;
[MDout missionout] = MDupdate(MDin,currentMD,bestn(9));
```

B.2.2. `ASCUdefault.m`

The `ASCUdefault.m` function receives the following inputs from the `ASCUMCTS.m` function:

- `treenode`. This vector is the row in the `ASCUMCTS.m` tree that corresponds to the current unexplored node identified by the tree policy. It includes the current mission index in the working `MD` mission array.
- `state`. This vector is the state associated with the current node. It provides the most recent landing times for each platform, corresponding to the current node.
- `MD`. This array is the mission array updated to correspond with the current unexplored node.
- `platform`. This array is the platform data array inputted by the user.
- `perform`. This array is the performance data array inputted by the user.

The function returns the following outputs:

- `y`. This value is the total mission reward resulting after randomly assigning available platforms to all missions remaining missions from the current unexplored node to the end of the mission array.

The function iterates through the following process:

1. Compute the distance and available platforms for the current mission. Available platforms include any platform that, having completed its previous mission assignment, has had time to land, reset, and can take-off to fulfill *any* part of the current mission requirement.
2. Randomly select one of the available platforms, or none at all.
3. Compute the reward and schedule for the assignment.
4. Update the `state` vector to include the new assignment schedule.
5. Update the total reward value and `treenode` for the current decision path.
6. Update the mission array to account for the current mission assignment.

7. Step the current mission and `treenode` index values (`mission` and `node`).

The function terminates and returns the total reward once it reaches the end of the mission array.

The code for `ASCUdefault.m` is below.

ASCUdefault.m

```
function y = ASCUdefault(treenode,state,MD,platform,perform)
%function ASCUdefault
%Written by Chris Marks
mission = treenode(2);
tree = treenode;
node = 1;
while mission <= size(MD,1)
  distance = sqrt(MD(mission,4)^2 + MD(mission,5)^2);
  traveltimes = (distance)./platform(:,2);
  arrivals = max(MD(mission,2)-traveltimes,state(:,node)+platform(:,3))+traveltimes;
  availset = find((2*traveltimes<platform(:,1))&(arrivals <
MD(mission,2)+MD(mission,3))&perform(:,MD(mission,1))>0)'; %available platforms.
  index = randi(length(availset)+1); %pick one at random.
  if (index == length(availset)+1)
    reward = 0;
    tree(node+1,:)  = [mission 0 0 0 0 tree(node,6)+reward MD(mission,1) 0 0]; %updated "tree" (more of an
assignment list in default algorithm)
    if mission < size(MD,1) %Non terminal.
      state(:,node+1) = state(:,node); %updated platform availability matrix.
    end
    [MD mission] = MDupdate(MD,mission,0);
    tree(node+1,2)=mission;
    node = node+1;
  else
    avail = availset(index); %Designate platform
    traveltime=traveltimes(avail); %Compute travel time
    timeon = min(MD(mission,2)+MD(mission,3)-arrivals(avail),platform(avail,1)-2*traveltimes(avail)); %
compute time on.
%    reward = (timeon/MD(mission,3))*MD(mission,6)*perform(avail,MD(mission,1)); %compute reward.
    reward = (timeon/1)*MD(mission,6)*perform(avail,MD(mission,1)); %compute reward.
    tree(node+1,:)  = [mission mission+1 avail 0 0 tree(node,6)+reward,MD(mission,1) 0 0]; %updated "tree"
(more of an assignment list in default algorithm)
    if mission < size(MD,1)
      state(:,node+1) = state(:,node); %updated platform availability matrix.
      state(avail,node+1) = min(MD(mission,2)+timeon+traveltime,MD(mission,2)+platform(avail,1)-traveltime); %
update platform landing times.
    end
    [MD mission] = MDupdate(MD,mission,arrivals(avail)+timeon);
    tree(node+1,2)=mission;
    node = node+1;
  end
end
y = tree(node-1,6);
```

B.2.3. `createchilds.m`

The `createchilds.m` function receives the following inputs:

- `MDmission`. This vector contains the current mission data from the mission array (`MD` in `ASCUMCTS.m`).
- `platform`. This array is the platform data inputted by the user.
- `nodestate`. This array gives the state information, consisting of most recent landing times for each platform, associated with the unexplored parent node.
- `treenode`. This vector gives the node vector from the `ASCUMCTS.m` tree that corresponds to the unexplored parent node.
- `node`. This value is the current unexplored parent node index.
- `mission`. This is the current mission index in the current mission array.
- `perform`. This is the platform performance data entered by the user.

The function returns the following outputs:

- `newnodes`. An array of new node vectors to add to the `ASCUMCTS.m` tree.
- `newstate`. An array of new states, consisting of vectors of landing times, corresponding to each of the new nodes.

The function finds all of the available platforms to accomplish the current mission based on the state platform availability information and on the mission information. For each available aircraft, the function computes:

1. The travel time to the mission area.
2. The take-off, mission area arrival, mission area departure, and landing times associated with assigning the platform to the current (parent node) mission.
3. The reward associated with the assignment.

This information is stored in a node corresponding to each available aircraft, and the landing times are also stored in a new state vector corresponding to each new node.

After producing a new node for each available platform, the function returns all of these new nodes, along with their corresponding states.

The code for `createchilds.m` is below.

`createchilds.m`

```
function [newnodes newstate] = createchilds(MDmission,platform,nodestate,treenode,node,mission,perform)
%function createchilds
```

```
%Written by Chris Marks
      dist = sqrt(MDmission(4)^2 + MDmission(5)^2); %distance to mission
      icounter = 1; %index of new node
      for avail = 1:size(nodestate,1) %this line defines the action space.
        traveltime=dist/platform(avail,2);
        if (2*traveltime < platform(avail,1)) %If the platform can make it to the mission area.
          takeoff = max(MDmission(2)-traveltime,nodestate(avail)+platform(avail,3));
          arrivetime = takeoff + traveltime;
          if (arrivetime < MDmission(2)+MDmission(3)) %if the platform can make it before the mission closes.
            departtime = min(MDmission(2) + MDmission(3),arrivetime + platform(avail,1)-2*traveltime);
            landtime = departtime + traveltime;
            timeon = departtime-arrivetime;
            if timeon <0
              sprintf('timeon<0')
            end
%            reward = (timeon/MDmission(3))*MDmission(6)*perform(avail,MDmission(1)); %Immediate reward for a
particular action
            reward = (timeon/1)*MDmission(6)*perform(avail,MDmission(1)); %Immediate reward for a particular
action
            newnodes(icounter,:)  = [node mission+1 avail 0 0 treenode(6)+reward MDmission(1) arrivetime
departtime]; %Save the new node.
%             if mission  = size(MD,1) %Do not want this in
%             order to preserve state-node integrity.
              newstate(:,icounter) = nodestate;
              newstate(avail,icounter) = landtime;
%            end
            icounter = icounter + 1;
          end
        end
      end
      %Create the no assignment child.
      newnodes(icounter,:)  = [node mission+1 0 0 0 treenode(6) MDmission(1) 0 0];
%      if mission  = size(MD,1) %We continue to track newstates in
%      order to preserve node-state integrity.
        newstate(:,icounter) = nodestate;
%      end
```

B.2.4. `MDupdate.m`

The `MDupdate.m` function receives the following inputs:

- `MD`. This array is the current mission array.
- `mission`. This value is the index of the current mission in the current mission array.
- `depart`. This value is the time when the platform assigned to complete the current mission departs the mission area.

The function returns the following outputs:

- `MDout`. This array is the updated mission array, based on the assigned platform's departure time.
- `missionout`. This value is the index of the current mission for the next iteration.

This function essentially adds any uncompleted part of any mission back into the mission array, starting at the time when the assigned platform departs. If no platform is assigned, it leaves a portion of the mission unfilled but re-inserts it for consideration immediately following the next mission in the current mission array (`MD`) that has a later start time.

The code for `MDupdate.m` is below.

`MDupdate.m`

```
function [MDout missionout] = MDupdate(MD,mission,depart)
%function MDupdate
%Written by Chris Marks
missionend = MD(mission,2) + MD(mission,3);
%can make the new mission when aircraft departs, when an aircraft becomes
%available, or when the next mission starts.
if depart == 0 %This means noassign.
  MDout = MD;
  n = 1;
  while (mission+n <= size(MD,1))&&(MD(mission,2)==MD(mission+n,2))
    n = n+1;
  end
  if ((mission + n <= size(MD,1))&&(MD(mission+1,2)<MD(mission,2)+MD(mission,3)))
    MDout(mission:mission+n-1,:)  = MD(mission+1:mission+n,:);
    MDout(mission+n,:)  = MD(mission,:);
    MDout(mission+n,[2 3]) = [MD(mission+n,2) missionend-MD(mission+n,2)];
    missionout = mission;
  else
    missionout = mission+1;
  end
else %There is an assignment
  if (mission  = size(MD,1))&&(depart < (MD(mission,2)+MD(mission,3)))
    n = mission+1;
    while ((n<=size(MD,1))&&(MD(n,2)<depart))
      n = n+1;
    end
    MDout(1:n-1,:)  = MD(1:n-1,:);
    MDout(n,:)  = MD(mission,:);
    MDout(n,[2 3]) = [depart MD(mission,3)-(depart-MD(mission,2))];
    MDout(n+1:size(MD,1)+1,:)  = MD(n:end,:);
    missionout = mission + 1;
  else
    MDout = MD;
    missionout = mission + 1;
  end
end
```

B.3. Analysis and Documentation Functions

This section provides brief documentation of the analysis and documentation functions that, while not part of the MCTS algorithm execution, provided for quick, insightful analysis of the results.

B.3.1. `missionplot.m`

The `missionplot.m` function receives the following inputs:

- `MDin`. This array is the original mission array input from the user.
- `scale`. This vector provides the minimum and maximum values for the x and y axes in the plot.

The function outputs a plot with time on the horizontal axis and mission value on the vertical axis. Each mission in the mission array is plotted as a horizontal red line depicting the time period when the mission is "open." Each red line is plotted at the height corresponding to the represented mission's value. The function plots the corresponding mission number from the user input mission array over each mission's red line. The function uses the scale to calculate number positions on the plot.

These plots serve as a an underlay for all of the assignment plots, including those in figures 4.3, 4.4, 4.5, 4.7, 4.8, 4.9, and 4.10.

The code for `missionplot.m` is below.

`missionplot.m`

```
function z = missionplot(MDin,scale)
%function missionplot
%Written by Chris Marks
lwidth = 2;
lcolor = 'r'; %'k' -> Black,'r'-> Red
x = [MDin(:,2) MDin(:,2)+MDin(:,3)];
y = [MDin(:,6) MDin(:,6)];
z=line(x',y','Color',lcolor,'LineWidth',lwidth);
% hoffset = 1.75/26 * (scale(2)-scale(1));
% voffset = 2.5/105 * (scale(4)-scale(3));
% for n = 1:size(MDin,1)
%   text((MDin(n,2) + MDin(n,3)/2 - hoffset), MDin(n,6) + voffset,['Mission ', int2str(MDin(n,1))]);
% end
hoffset = 0/26 * (scale(2)-scale(1));
voffset = 2.5/105 * (scale(4)-scale(3));
for n = 1:size(MDin,1)
  text((MDin(n,2) + MDin(n,3)/2 - hoffset), MDin(n,6) + voffset,
int2str(MDin(n,1)),'HorizontalAlignment','center');
end
```

B.3.2. `allassignmentplot.m`

The `allassignmentplot.m` function receives the following inputs:

- `assignments`. This array is a complete assignment schedule output coming from the MCTS algorithm.

- `MDin`. This is the original mission array input from the user.

This function returns a plot depicting assignments for all platforms. It is meant to be overlaid on a mission plot that is already produced. For an example of this plot, see figure 4.3. Figure B.1 shows the output of this function when there are many platforms.

The code for `allassignmentplot.m` is below.

`allassignmentplot.m`

```
function z = allassignmentplot(assignments,MDin)
%function allassignmentplot
%Written by Chris Marks
lwidth = 1.25;
lcolor = 'k'; %'k' -> Black,'r'-> Red
index = 1;
for j = 1:size(assignments,1)
  if(assignments(j,1)  = 0)
    xx(index,:)  = assignments(j,3:6);
    m = assignments(j,2);
    yy(index,:)  = [0 MDin(m,6) MDin(m,6) 0];
    index = index + 1;
  end
end
z=line(xx',yy','Color',lcolor,'LineWidth',lwidth);
```

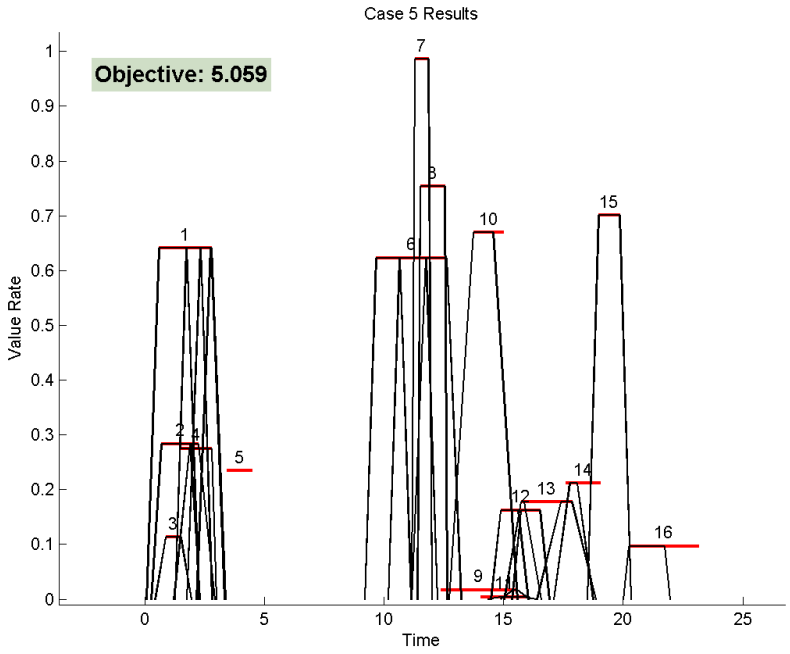

Figure B.1: Example `allassignmentsplot.m` output for multiple platforms.

B.3.3. `assignmentplot.m`

The `assignmentplot.m` function is similar to `allassignmentplot.m` except that the resulting plot does not display lines representing platforms' travels to and from the missions. Figures 4.8, 4.9, and 4.10 are all outputs from this function.

The code for `assignmentplot.m` is below.

`assignmentplot.m`

```
function z = assignmentplot(assignments,MDin)
%function assignmentplot
%Written by Chris Marks
lwidth = 2;
lcolor = 'k'; %'k' -> Black,'r'-> Red
index = 1;
for j = 1:size(assignments,1)
  if(assignments(j,1)  = 0)
    xx(index,:)  = assignments(j,4:5);
    m = assignments(j,2);
    yy(index,:)  = [MDin(m,6) MDin(m,6)];
    index = index + 1;
  end
end
z=line(xx',yy','Color',lcolor,'LineWidth',lwidth);
```

B.3.4. `singleassignmentplot.m`

The `singleassignmentplot.m` function receives the following inputs:

- `assignments`. This array is a complete assignment schedule output coming from the MCTS algorithm.
- `MDin`. This is the original mission array input from the user.
- `platform`. This value is the index of the platform of interest in the platform input array.

The function returns a plot very similar to that returned by the `allassignmentsplot.m` function, except that instead of plotting the schedule for all of the platforms, it only depicts the schedule for the input platform. For example, the single assignment plot for platform 7, coming from the same scenario (case 5) used to generate figure B.1, is depicted in figure B.2.

The code for `singleassignmentplot.m` is below.

`singleassignmentplot.m`

```
function z = singleassignmentplot(assignments,MDin,platform)
```

```
%function singleassignmentplot
%Written by Chris Marks
lwidth = 1.25;
lcolor = 'k'; %'k' -> Black,'r'-> Red
index = 1;
for j = 1:size(assignments,1)
  if(assignments(j,1)==platform)
    xx(index,:)  = assignments(j,3:6);
    m = assignments(j,2);
    yy(index,:)  = [0 MDin(m,6) MDin(m,6) 0];
    index = index + 1;
  end
end
z=line(xx',yy','Color',lcolor,'LineWidth',lwidth);
```

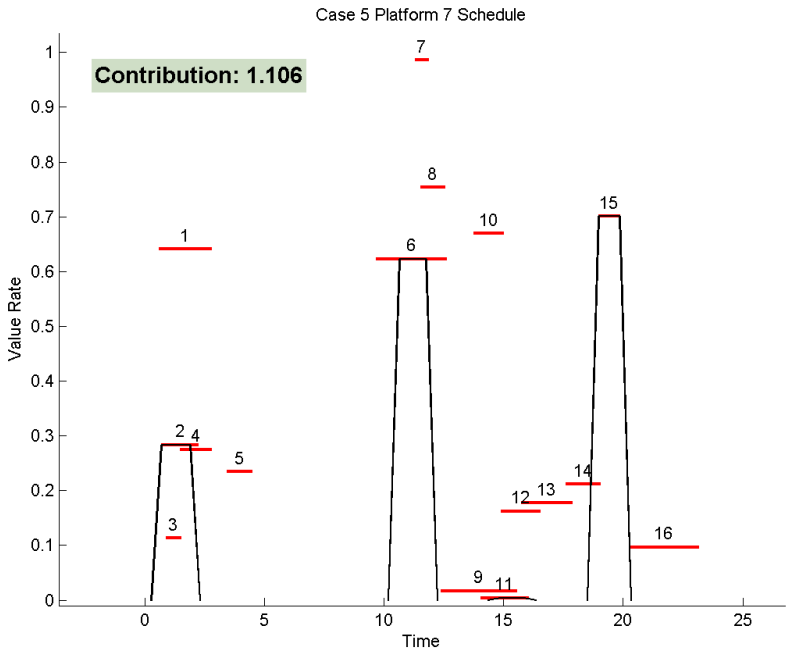

Figure B.2: Example `singleassignmentsplot.m` output for platform 7 in case 5.

B.3.5. `document.m`

The `document.m` function receives the following inputs:

- `x`. This value is the number corresponding to the test case.
- `assignments`. This array is a complete assignment schedule output coming from the MCTS algorithm.

The function also reads in the three use input arrays pertaining to the test case:

- `platform`. The platform data array.
- `MDin`. The mission data array.
- `perform`. The performance data array.

The function uses these inputs to write a basic LaTeXinput file that documents the inputs for the test case and the results. The three user input arrays are formatted in tables, the `assignments.m` or `allassignments.m` plot is imported as a figure, and the assignment schedule is formatted as a table. This output makes it easier to import the results into a final report format.

The code for `document.m` is below.

`document.m`

```
function y = document(x,assignments)
%function document
%Written by Chris Marks
scenario = x;
MDin = xlsread(sprintf('../case%u/input%u.xlsx',scenario,scenario),'MD'); %List missions in order.
platform = xlsread(sprintf('../case%u/input%u.xlsx',scenario,scenario),'platform');
perform = xlsread(sprintf('../case%u/input%u.xlsx',scenario,scenario),'perform');
fid = fopen(sprintf('../case%u/readme.tex',scenario),'w');
fprintf(fid,'\\documentclassarticle\n\n');
fprintf(fid,'\\usepackagegraphicx\n');
%fprintf(fid,'\\usepackagelongtable\n');
fprintf(fid,'\n\n\\begindocument\n\n');
fprintf(fid,'\\sectionCase %u\n\n',scenario);
fprintf(fid,'\\subsectionInputs\n\n');
if size(platform,1)==1
  fprintf(fid,'In this case we have one platform with the characteristics shown in table \\refplatform%u,
below.\n\n',scenario);
else
  fprintf(fid,'In this case we have %u platforms with the characteristics shown in table \\refplatform%u,
below.\n\n',size(platform,1),scenario);
end
fprintf(fid,'\\begintable[!hbt]\n\\centering\n\\begintabularcccc \n');
fprintf(fid,'Platform & Endurance & Speed & Reset \\\\ \\hline \n');
platform =[(1:size(platform,1))' platform];
fprintf(fid,'%u & %4.2f & %6.2f & %4.2f \\\\ \n',platform');
fprintf(fid,'\\endtabular\n\\captionPlatform inputs for case
%u.\\labelplatform%u\n\\endtable\n\n',scenario,scenario);
if size(MDin,1)==1
  fprintf(fid,'In this case we have one mission with the characteristics shown in table \\refmission%u,
below.\n\n',scenario);
else
  fprintf(fid,'In this case we have %u missions with the characteristics shown in table \\refmission%u,
below.\n\n',size(platform,1),scenario);
end
fprintf(fid,'\\begintable[!hbt]\n\\centering\n\\begintabularcccccc \n');
fprintf(fid,'Mission \\# & Start time & Duration & $x$ coordinate & $y$ coordinate & Value rate \\\\ \\hline
\n');
fprintf(fid,'%u & %4.2f & %4.2f & %4.2f & %4.2f & %4.2f \\\\ \n',MDin');
fprintf(fid,'\\endtabular\n\\captionMission inputs for case
%u.\\labelmission%u\n\\endtable\n\n',scenario,scenario);
fprintf(fid,'Finally, table \\refperform%u provides each platform''s performance coefficient for each
mission.\n\n',scenario);
tablestr = blanks(size(perform,2)+2);
tablestr(1:2) = 'c|';
for i = 3:(size(perform,2)+2)
```

```
  tablestr(i)='c';
end
perform =[(1:size(perform,1))' perform];
fprintf(fid,'\\begintable[!hbt]\n\\centering\n\\begintabular%s \n',tablestr);
fprintf(fid,'Platform ')
for i = 2:size(perform,2)
  fprintf(fid,'& Mission \\#%u ',i-1);
end
fprintf(fid,' \\\\ \\hline \n');
for i = 1:size(perform,1)
  fprintf(fid, '%u ',perform(i,1));
  for j = 2:size(perform,2)
    fprintf(fid,'& %3.2f ',perform(i,j));
  end
  fprintf(fid,'\\\\ \n');
end
fprintf(fid,'\\endtabular\n\\captionPerformance inputs for case
%u.\\labelperform%u\n\\endtable\n\n',scenario,scenario);
fprintf(fid,'\n\n\\subsectionResults\n\n');
fprintf(fid,'The MCTS scheduling algorithm produced the schedule shown in table \\refassignments%u. Figure
\\refresults%u gives a visual depiction of this schedule.\n\n',scenario,scenario);
fprintf(fid,'\\begintable[!hbt]\n\\centering\n\\begintabularcccccc \n');
fprintf(fid,'Platform & Mission \\# & Take-off & Arrive & Depart & Land \\\\ \\hline \n');
fprintf(fid,'%u & %u & %4.2f & %4.2f & %4.2f & %4.2f \\\\ \n',assignments');
fprintf(fid,'\\endtabular\n\\captionSchedule produced for case
%u.\\labelassignments%u\n\\endtable\n\n',scenario,scenario);
fprintf(fid,'\\beginfigure[!hbt]\n\\centering\n');
fprintf(fid,'\\fbox\n\\includegraphics[width = 0.95\\textwidth]results%u.png \n \n',scenario);
fprintf(fid,'\\captionVisual depiction of schedule for case
%u.\\labelresults%u\n\\endfigure\n\n',scenario,scenario);
fprintf(fid,'\\enddocument');
fclose(fid);
```

This page intentionally left blank.

References

[1] Darryl K Ahner, Arnold H Buss, and John Ruck. Assignment scheduling capability for unmanned aerial vehicles: a discrete event simulation with optimization in the loop approach to solving a scheduling problem. In *Proceedings of the 38th conference on Winter simulation*, pages 1349–1356. Winter Simulation Conference, 2006.

[2] C.B. Browne, E. Powley, D. Whitehouse, S.M. Lucas, P.I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton. A survey of Monte Carlo tree search methods. *Computational Intelligence and AI in Games, IEEE Transactions on*, 4(1):1–43, march 2012. ISSN 1943-068X. doi: 10.1109/TCIAIG.2012.2186810.

[3] Arnold H Buss and Darryl K Ahner. Dynamic allocation of fires and sensors (DAFS): a low-resolution simulation for rapid modeling. In *Simulation Conference, 2006. WSC 06. Proceedings of the Winter*, pages 1357–1364. IEEE, 2006.

[4] Dr. Arnold Buss, MAJ Christopher Marks, MAJ Peter Nesbitt, and LTC Jonathan Alt. Modeling updates in support of armed aerial scout. Technical memorandum TRAC-M-TM-13-043, TRADOC Analysis Center, Monterey, 700 Dyer Road, Monterey, CA 93943-0692, 24 April 2013.

[5] John B Gilmer Jr and Frederick J Sullivan. Alternative implementations of multitrajectory simulation. In *Simulation Conference Proceedings, 1998. Winter*, volume 1, pages 865–872. IEEE, 1998.

[6] John B Gilmer Jr and Frederick J Sullivan. Recursive simulation to aid models of decisionmaking. In *Simulation Conference, 2000. Proceedings. Winter*, volume 1, pages 958–963. IEEE, 2000.

[7] Headquarters, Department of the Army. *FM 3-0: Operations*. Government Printing Office.

[8] Volodymyr Kuleshov and Doina Precup. Algorithms for the multi-armed bandit problem. *Journal of Machine Learning*, 2010.

[9] Kyle Y. Lin. New results on a stochastic duel game with each force consisting of heterogeneous units. Technical Report NPS-OR-13-002, Naval Postgraduate School, Monterey, CA 93943-0692, February 2013. URL `http://calhoun.nps.edu/public/handle/10945/30315`.

[10] MAJ Edward Masotti. Infantry at risk—vignette 3, combat outpost (COP) attack. Powerpoint presentation, Training and Doctrine Command Analysis Center—Monterey (TRAC-MTRY), 700 Dyer Road, Monterey, CA 93943, September 2012.

[11] Dave Ohman. COMBATXXI, defined. COMBAT XXI online documentation, Training and Doctrine Command Analysis Center—White Sands Missile Range (TRAC-WSMR), Martin Luther King Drive, White Sands Missile Range, NM 88002-5502, 28 September 2011.